Aportaciones Teóricas y Prácticas a las Tecnologías del Habla

SAUTRELA

Un Entorno de Desarrollo

Tesis Doctoral

Mikel Peñagarikano Badiola

Enero 2016

Director:
Dr. Germán Bordel

Elektrizitatea eta Elektronika Saila
Departamento de Electricidad y Electrónica

ZTF-FCT
Zientzia eta Teknologia Fakultatea
Facultad de Ciencia y Tecnología

eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea

Departamento de Electricidad y Electrónica
Elektrizitatea eta Elektronika Saila

ZTF-FCT
Zientzia eta Teknologia Fakultatea
Facultad de Ciencia y Tecnología

eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea

Aportaciones Teóricas y Prácticas a las Tecnologías del Habla: Sautrela, un Entorno de Desarrollo

Autor:
Mikel Peñagarikano Badiola

Director:
Germán Bordel García

Memoria que se presenta para optar al grado de

Doctor en Ciencias

Leioa, 2 de diciembre de 2015

Universidad del País Vasco

Resumen

Facultad de Ciencia y Tecnología
Departamento de Electricidad y Electrónica

Tesis Doctoral

Aportaciones Teóricas y Prácticas a las Tecnologías del Habla: Sautrela, un Entorno de Desarrollo

por Mikel Peñagarikano Badiola

Las Tecnologías del Habla (TH) abarcan un amplio conjunto de áreas de investigación: el reconocimiento del habla, la lengua o el locutor, la síntesis del habla, la interacción multimodal, la recuperación de información en recursos multimedia, etc. Sin embargo, gran parte de estas disciplinas carece de las herramientas software que permitan implementar de una manera sencilla nuevas metodologías o diseños. El investigador se convierte así en un programador que debe modificar una y otra vez su herramienta de trabajo. Debido tanto a la complejidad que supone como a los problemas que acarrea la modificación de software de terceros, en los grupos de investigación surge a menudo la necesidad de crear un software propio que ofrezca un mayor conocimiento y control sobre la herramienta de trabajo.

La presente memoria detalla la estructura de Sautrela, con las decisiones de diseño y las aportaciones que incorpora a las TH, explicando tanto sus bases teóricas como los aspectos propios de su implementación. El entorno, que ha sido desarrollado íntegramente por el aspirante, ha sido tanto un objetivo en sí mismo como un elemento de utilidad en toda su trayectoria investigadora, que abarca numerosos trabajos resumidos igualmente en este documento. Sautrela es además una herramienta clave para la actividad del Grupo de Trabajo en Tecnologías Software, del que el aspirante fue cofundador, y se ofrece como software de código abierto a la comunidad científica. Esta herramienta define un conjunto versátil y extensible de componentes orientado al desarrollo de sistemas de reconocimiento de patrones, y especialmente enfocado a las TH. Su naturaleza modular y altamente configurable ofrece al investigador la oportunidad de abordar un gran número de problemas sin necesidad de añadir una sola línea de código. Sautrela es también un entorno extensible que permite integrar de manera sencilla nuevos componentes desarrollados por terceros.

Euskal Herriko Unibertsitatea

Laburpena

Zientzia eta Teknologia Fakultatea
Elektrizitate eta Elektronika Saila

Tesi Doktorala

Mintza-Teknologiei Ekarpen Teoriko eta Praktikoak: Sautrela, Garapen-Ingurune bat

Mikel Peñagarikano Badiola

Mintzo Teknologiek ezagutza arlo ezberdinak hartzen dituzte beren baitan: mintzo, hizkuntz eta hiztun ezagutze automatikoa, mintzoaren sintesia, interakzio multimodala, informazio-berreskuratzea multimedia baliabideetan, etab. Arlo hauetako gehienetan, ordea, metodologia berriak frogatzeko aukera erraz bat eskeini ahal duten software tresnen gabezia izan ohi dugu maiz. Hala, ikertzailea, bere software lan tresnak behin eta berriro eraldatzea behartua izango da, behe mailako programazio lanetara punitua. Besteen softwarea aldatzeak suposatzen duen konplexutasuna eta dakartzan arazoak direla eta, ikerketa taldeetan euren berezko softwarea garatzeko beharra sortzen da maiz. Izan ere, berezko softwareak, garapenaren ezagutza eta kontrol egokiagoak eskeintzen ditu.

Oroitza-txosten honek, Mintzo Teknologietara bideratutako software ingurune bat du ardatz: Sautrela. Doktoregaia izan da software ingurune honen sustatzaile eta garatzilea, bere ikerketa ibilbidean zehar (txosten honetan ere laburki azaldua dena) erabilpen zabala izan duelarik. Doktoregaia sortzaile izan zeneko Software Teknologien Lantaldearen jardueraren tresna adierazgarria ere bada Sautrela. Tresna hau, Mintzo Teknologietara bideratutako eta eredu-ezagutze sistemak sortzeko gai den erabilera anitzeko osagai-multzo hedagarri batez osotzen da. Modularra eta guztiz konfiguragarria izaki, kode lerro bakarra gehitu beharrik izan gabe problema ugariri aurre egiteko aukera eskeintzen dio ikertzaileari. Sautrela ingurune hedagarria ere bada: besteek garatutako software osagaiak erraz integratu ahal dira, halaber.

University of the Basque Country

Abstract

Faculty of Science and Technology
Department of Electricity and Electronics

Doctoral Thesis

Theoretical and Practical Contributions to the Speech Technologies: Sautrela, a Development Environment

by Mikel Peñagarikano Badiola

Speech Technologies include a broad range of research areas: speech, language or speaker recognition, speech synthesis, multimodal interaction, multimedia information retrieval, etc. However, many of these disciplines lack of software tools that allow a simple way to implement new methodologies or designs. The researcher thus becomes a low level programmer who must periodically modify software tools. Due to both its complexity and the problems involved in third-party software modification, the need to create their own software tools (providing greater insight and control) arises frequently in the research groups.

This thesis focuses on the design and practical use of a Speech Technologies oriented software development environment: Sautrela. This toolkit has been completely developed by the applicant, being a useful element throughout his research career, which is also summarized in this document. Sautrela is also a key tool for the activity of the Software Technology Working Group, which was co-founded by the applicant. This tool defines a versatile and extensible set of components aimed at developing systems for pattern recognition, especially focused on Speech Technologies. Its modular and configurable nature offers the researcher the opportunity to address a large number of problems without the need to add a single line of code. Sautrela is also an extensible environment that can easily integrate third-party software components.

A mis padres,

Ana y Juan Luis

Debile principium melior fortuna sequatur

— Bernat Etxepare, *Linguae Vasconum Primitiae* (Bordele, 1545)

Agradecimientos

Deseo expresar mi más sincero agradecimiento a todas aquellas personas que me han ayudado y acompañado en este largo viaje.

En primer lugar, y de manera especial, a Germán, mi mentor, director y compañero, por todos estos años en los que tanto he aprendido, y por ser el germen de este grupo de investigación en el que he disfrutado y he podido desarrollarme profesionalmente. También al resto de integrantes del Grupo de Investigación GTTS. A Luisja, por haber apostado por nuestro proyecto, y por todo lo aportado en esas acaloradas reuniones de *lluvia de ideas* que tanto nos gustan, y cómo no, por todo el apoyo recibido en la redacción de esta memoria. A Amparo, por su comprensión y constancia, por motivarnos y ayudarnos a remar en una misma dirección. Y a Mireia, por haber sufrido estoicamente los constantes cambios de Sautrela y por habernos enseñado que somos capaces de más. Ahora sí, podemos.

Me gustaría también agradecer a los miembros del Departamento de Electricidad y Electrónica de la Facultad de Ciencia y Tecnología de la UPV/EHU el apoyo recibido, así como celebrar el sentirme miembro de una gran familia.

No puedo olvidar a toda mi familia que siempre ha estado ahí. A mis hermanos y padres, que tanto han tenido que esperar hasta ver culminado este trabajo. Y, por supuesto, a mi mujer Ana y a mis hijas Ane y Nora, a las que no he podido dedicar todo el tiempo que desearía, y espero poder compensar.

A todos y todas, muchas gracias.

Glosario de acrónimos

ACC	**ACC**uracy
AIFC	**A**udio **I**nterchange **F**ile Format (**C**ompressed variant)
AIFF	**A**udio **I**nterchange **F**ile **F**ormat
ASCII	**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange
AU	**AU**dio file format (NeXT/Sun sound file)
awk	A programming language by Alfred **A**ho, Peter **W**einberger, and Brian **K**ernighan
BSD	**B**erkeley **S**oftware **D**istribution
CAS	**C**ompare **A**nd **S**wap
CDATA	**C**haracter **DATA**
CHMM	**C**ontinuous **H**idden **M**arkov **M**odel
CMN	**C**epstral **M**ean **N**ormalization
CRC	**C**yclic **R**edundancy **C**heck
DCT	**D**iscrete **C**osine **T**ransform
DTD	**D**ocument **T**ype **D**efinition
DWFSA	**D**eterministic **W**eighted **F**inite-**S**tate **A**utomata
eLBG	**e**nhanced **LBG**
EM	**E**xpectation–**M**aximization
ER	**E**rror **R**ate
FFT	**F**ast **F**ourier **T**ransform
FIFO	**F**irst **I**n **F**irst **O**ut
GLDS	**G**eneralized **L**inear **D**iscriminant **S**equence kernel
GMM	**G**aussian **M**ixture **M**odel
GNU	**G**NU's **N**ot **U**nix
grep	**G**lobally search a **R**egular **E**xpression and **P**rint
GUI	**G**raphical **U**ser **I**nterface

HMM	**H**idden **M**arkov **M**odel
HTK	**H**idden Markov Model **T**ool**K**it
IIR	**I**nfinite **I**mpulse **R**esponse
ISO	**I**nternational **S**tandards **O**rganization
JAR	**J**ava **AR**chive
JVM	**J**ava **V**irtual **M**achine
KTLSS	**K**-**T**estable **L**anguage in the **S**trict **S**ense
LGB	**L**indo **B**uzo **G**rey (clustering algorithm)
LMM	**L**ayered **M**arkov **M**odel
MAP	**M**aximum **A** **P**osteriori
MFCC	**M**el-**F**requency **C**epstral **C**oefficients
ML	**M**aximum **L**ikelihood
MLE	**M**aximum **L**ikelihood **E**stimation
NdWFSA	**N**on-**d**eterministic **W**eighted **F**inite-**S**tate **A**utomata
PCM	**P**ulse-**C**ode **M**odulation
Perl	**P**ractical **E**xtraction and **R**eport **L**anguage
PHP	**P**ersonal **H**ome **P**age (Hypertext Preprocessor)
PLP	**P**erceptual **L**inear **P**rediction
RAH	**R**econocimiento **A**utomático del **H**abla
RASTA	**R**el**A**tive **S**pec**T**r**A**
SDC	**S**hifted **D**elta **C**epstrum
SGML	**S**tandard **G**eneralized **M**arkup **L**anguage
SND	NeXT **S**ou**ND** audio file format
SVM	**S**upport **V**ector **M**achine
Tcl	**T**ool **C**ommand **L**anguage
TH	**T**ecnologías del **H**abla
UBM	**U**niversal **B**ackground **M**odel
URLs	**U**niform **R**esource **L**ocator
UTF	**U**nicode **T**ransformation **F**ormat
UTF-16	**16**-bit **U**nicode **T**ransformation **F**ormat
UTF-32	**32**-bit **U**nicode **T**ransformation **F**ormat
UTF-8	**8**-bit **U**nicode **T**ransformation **F**ormat
W3C	**W**orld **W**ide **W**eb **C**onsortium

WAVE	**WAVE**form audio file format
WFSA	**W**eighted **F**inite-**S**tate **A**utomata
WFSASet	**W**eighted **F**inite-**S**tate **A**utomata **Set**
WWW	**W**orld **W**ide **W**eb
XML	e**X**tensible **M**arkup **L**anguage

Glosario de notación en pseudocódigo

Los algoritmos descritos en la presente memoria se han beneficiado de una notación matemática que simplifica en lo posible su descripción en pseudocódigo. El presente glosario pretende definir la notación utilizada, que está agrupada en tres apartados: la notación relativa a conjuntos, secuencias o tuplas, y mapas.

Conjuntos

Un conjunto representa una colección de elementos no repetidos.

$\{\,\}$	Conjunto vacío.
$\{a, b, c, d\}$	Conjunto compuesto por los elementos a, b, c y d.
$\{a_1, a_2, \ldots, a_n\}$	Conjunto compuesto por los elementos a_1, a_2, hasta a_n.
$\lvert A\rvert$	Cardinal (número de elementos) del conjunto A.
$A \cup B$	Unión de los conjuntos A y B.
$A \cup \{b\}$	Conjunto que resulta de la unión del conjunto A y el conjunto que contiene un único elemento b (el conjunto A no es modificado).
if $x \in A$ **then** | ... **end**	Evalúa las sentencias del bloque **if** si el elemento x está contenido en el conjunto A.
for $x \in A$ **do** | ... **end**	Recorre los elementos del conjunto A, evaluando las sentencias del bloque **for** para cada uno de los elementos x contenidos en A.
$\arg\max_{x \in A} \{f(x)\}$	Retorna un elemento $x \in A$ que maximiza $f(x)$, de tal manera que $\forall\, y \in A,\ f(x) \geq f(y)$

Secuencias o tuplas

Una secuencia o tupla (en adelante se utilizará el término secuencia) representa una colección ordenada de elementos.

$( )$	Secuencia vacía.
(a, b, c, d)	Secuencia compuesta por los elementos a, b, c y d.
$(a_1, a_2, \ldots, a_n)$	Secuencia compuesta por los elementos a_1, a_2, hasta a_n.
$\|A\|$	Cardinal (número de elementos) de la secuencia A.
$(x, y, z) \leftarrow (a, b, c)$	Asigna los valores a, b y c de la secuencia (a, b, c) a las variables x, y y z, respectivamente.
$(x, y, z) \leftarrow t$	La variable $t = (a, b, c)$ debe ser una secuencia de tres elementos. Equivale a**:** $(x, y, z) \leftarrow (a, b, c)$
$(x, \sim, z) \leftarrow (a, b, c)$	Asigna los valores a, y c a las variables x y z. El valor b es descartado.
$A + B$	Concatenación de las secuencias A y B.
$A + (b)$	Secuencia que resulta de la concatenación de la secuencia A y la secuencia que contiene un único elemento b (la secuencia A no es modificada).
$reversed\,(A)$	Retorna una secuencia en orden inverso a A.
if $x \in A$ **then** | ... **end**	Evalúa las sentencias del bloque **if** si el elemento x está contenido en la secuancia o tupla A.
for $x \in A$ **do** | ... **end**	Recorre los elementos de la secuencia A en el orden natural de la secuencia, evaluando las sentencias del bloque **for** para cada uno de los elementos x contenidos en A.
for $(a, b, c) \in A$ **do** | ... **end**	Recorre los elementos del conjunto o secuencia A. Los elementos deben corresponder con secuencias de tres valores. Equivale a **for** $x \in A$ **do** | $(a, b, c) \leftarrow x$ | ... **end**

for $(\sim, b, c) \in A$ **do**
| ...
end

Recorre los elementos del conjunto o secuencia A. Los elementos deben corresponder con secuencias de tres valores, descartándose el primero de ellos. Equivale a

```
for x ∈ A do
|   (∼, b, c) ← x
|   ...
end
```

$\arg\max_{x \in A} \{f(x)\}$

Retorna un elemento $x \in A$ que maximiza $f(x)$, de tal manera que $\forall\, y \in A,\ f(x) \geq f(y)$

Mapas

Un mapa, también denominado vector asociativo o diccionario, está compuesto por un conjunto de claves asociadas a una colección de valores. Cada par *clave* $\rightarrow$ *valor* es denominado entrada. Dos entradas pueden tener un mismo valor y claves diferentes, pero no se puede dar el caso contrario. El valor `NIL` representa el valor nulo, e indica la no existencia del valor consultado.

$\{\mapsto\}$

Mapa vacío.

$\{k \mapsto v\}$

Mapa compuesto por una única entrada cuya clave y valor quedan definidos por la clave k y el valor v.

$\{k_1 \mapsto v_1\ ,\ k_2 \mapsto v_2\}$

Mapa compuesto por las entradas $k_1 \mapsto v_1$ y $k_2 \mapsto v_2$

$|M|$

Cardinal (número de entradas) del mapa M.

$M[k]$

Retorna: $\begin{cases} v, & \text{si } \exists\, k \mapsto v \in M \\ \texttt{NIL}, & \text{en caso contrario} \end{cases}$

$M[k] \leftarrow v$

Añade la entrada $k \mapsto v$ al mapa M. Si el mapa M contenía previamente otra entrada $k \mapsto w$, esta es eliminada.

for $k \mapsto v \in M$ **do**
| ...
end

Recorre las entradas del mapa M, evaluando las sentencias del bloque **for** para cada una de las entradas. Las variables k y v contendrán la clave y el valor de cada una de las entradas, respectivamente.

for $k \mapsto (v_1, v_2) \in M$ **do**
| ...
end

Recorre las entradas del mapa M, evaluando las sentencias del bloque **for** para cada una de las entradas. Todos los valores del mapa deben corresponder con tuplas de dos elementos. Equivale a**:**

for $k \mapsto v \in M$ **do**
| $(v_1, v_2) \leftarrow v$
| ...
end

$\arg\max_{k \mapsto v \in M} \{f(k, v)\}$

Retorna la tupla (k, v) compuesta por la clave y el valor de la entrada $k \mapsto v \in M$ que maximiza $f(x, y)$, de tal manera que $\forall\, a \mapsto b \in M,\ f(k, v) \geq f(a, b)$.

$\arg\max_{k \mapsto (v_1, v_2) \in M} \{f(k, x, y)\}$

Retorna la tupla $(k, (v_1, v_2))$ compuesta por la clave y el valor de la entrada $k \mapsto v = (v_1, v_2) \in M$ que maximiza $f(x, y, z)$, de tal manera que $\forall\, a \mapsto (b_1, b_2) \in M,\ f(k, v_1, v_2) \geq f(a, b_1, b_2)$. Todos los valores del mapa deben corresponder con tuplas de dos elementos.

Índice general

Índice de figuras

Índice de listados

Índice de tablas

Índice de algoritmos

Capítulo 1

Introducción

1.1. Contexto, motivación y objetivos

Las Tecnologías del Habla (TH) abarcan un amplio conjunto de áreas de investigación entre las que podemos encontrar el reconocimiento del habla, la lengua o el locutor, la síntesis del habla, la interacción multimodal o la recuperación de información en recursos multimedia. A menudo los equipos o grupos de investigación de esta área suelen enfocar su trabajo en más de una de estas disciplinas, bien por la afinidad inherente entre algunas de ellas, bien por la búsqueda de nuevos retos que pudieran resultar más atractivos, o simplemente por la deriva de las fuentes de financiación.

Sin embargo en la mayor parte de estas disciplinas nos encontramos con dos grandes dificultades para el desarrollo de la investigación: el acceso a los datos y la disponibilidad de herramientas software adecuadas. Para poder contrastar cualquier idea innovadora es preciso contar con una o más bases de datos frente a las cuales poder verificar la validez de la propuesta. Dichas bases de datos son a menudo excesivamente costosas, estando al alcance únicamente de aquellos grupos de investigación que cuenten con suficientes recursos. En algunos casos puede darse incluso la circunstancia de que alguna base de datos utilizada por la comunidad investigadora no cuente con acceso público, estando su uso restringido a aquellos grupos que hayan tomado parte en alguna iniciativa previa (a su vez con acceso libre o restringido).

El acceso a las bases de datos, no obstante, no es sino el menor de los inconvenientes. Un investigador de las TH cuenta básicamente con dos tipos de herramientas de trabajo: las herramientas hardware o servidores de cómputo y el conjunto de herramientas software con las que implementar las tecnologías estudiadas. En lo que respecta al hardware, el depender únicamente de plataformas genéricas de cómputo (ordenadores) permite un acceso sencillo a dichas herramientas (si lo comparamos con otras disciplinas que precisan de equipamiento específico). En lo que respecta a las herramientas software, sin embargo, podemos afirmar que no existen conjuntos de herramientas que permitan implementar de una manera sencilla nuevas metodologías o diseños. El investigador se convierte así en un programador que debe modificar una y otra vez su herramienta de trabajo.

Debido tanto a la complejidad que supone como a los problemas que acarrea la modificación de software de terceros, a menudo surge en los grupos de investigación la necesidad de crear un software propio que ofrezca un mayor conocimiento y control sobre la herramienta de trabajo. Sin embargo, la complejidad actual de dichos sistemas hace a menudo inviable desarrollar tecnología propia de vanguardia.

La presente memoria se centra en el diseño y uso práctico de un entorno software de aplicación a las TH: Sautrela. Este entorno ha sido íntegramente desarrollado por el aspirante, siendo un elemento de utilidad en toda su trayectoria investigadora, que se resume igualmente en este documento. Sautrela es además una herramienta clave para la actividad del Grupo de Trabajo en Tecnologías Software, del que el aspirante fue cofundador. Esta herramienta define un conjunto versátil y extensible de componentes orientado al desarrollo de sistemas de reconocimiento de secuencias de patrones, y especialmente enfocado a las TH. Su naturaleza modular y altamente configurable ofrece al investigador la posibilidad de abordar un gran número de problemas sin necesidad de añadir una sola línea de código. Sautrela es también un entorno extensible que permite integrar de manera sencilla nuevos componentes desarrollados por terceros.

1.2. Estructura de la memoria

La presente memoria consta de siete capítulos y viene acompañada de tres apéndices. El presente capítulo analiza la motivación y los objetivos del trabajo acometido, e introduce algunas de las características del entorno de desarrollo Sautrela.

El Capítulo 2 analiza el estado del arte en lo que respecta a herramientas software de uso común en las tecnologías del habla, describiendo las principales características de algunos entornos que han sido considerados de relevancia, tales como HTK, Sphinx y Kaldi. También se presenta el entorno de desarrollo Sautrela, analizando sus principales características.

El Capítulo 3 describe la arquitectura software de Sautrela, la cual define un entorno de desarrollo de ámbito general para el procesamiento de la información. A diferencia de otros entornos de desarrollo, Sautrela no es una herramienta específica diseñada para la generación de motores de reconocimiento, sino una arquitectura neutral y modular cuya finalidad es la concatenación de módulos de procesamiento.

En el Capítulo 4 se muestran los componentes o módulos que acompañan a Sautrela y que conforman un entorno de desarrollo para las Tecnologías del Habla. Dichos módulos ofrecen funcionalidades relacionadas con la entrada y salida de audio, el manejo de bases de datos acústicas, el procesamiento de la señal y el modelado (entrenamiento y decodificación basados en autómatas de estados finitos y mezclas de Gaussianas).

El Capítulo 5 analiza las aportaciones teóricas o algorítmicas relevantes contenidas en Sautrela y relacionadas con las Tecnologías del Habla. Dichas aportaciones están en su mayoría enmarcadas dentro de las técnicas de modelado utilizadas por Sautrela.

El Capítulo 6 contiene un resumen de la trayectoria investigadora del aspirante, que viene acompañado de las correspondientes contribuciones realizadas en cada una de las disciplinas en las que ha tomado parte.

En el Capítulo 7 se resumen las conclusiones del trabajo presentado y se indican las líneas principales de trabajo que se prevé abordar en el futuro.

Finalmente, la memoria concluye con tres apéndices. El Apéndice A muestra los pasos a seguir para la correcta instalación de Sautrela y aborda además cada una de las fases de la creación de un motor de reconocimiento de gran vocabulario: la preparación de una base de datos acústica, la creación y entrenamiento de modelos acústicos, léxicos y de lenguaje, y la ejecución y optimización del motor de reconocimiento. El Apéndice B contiene la documentación de todos los comandos de Sautrela. Por último, el Apéndice C contiene la información relativa a cada uno de los módulos de procesamiento.

Capítulo 2

Entornos de desarrollo en las Tecnologías del Habla

2.1. Introducción

Las Tecnologías del Habla cuentan con una amplia oferta de herramientas de código abierto que sirven a los investigadores como plataformas de desarrollo sobre las cuales estudiar ciertos aspectos específicos del área de conocimiento. En el ámbito concreto del Reconocimiento Automático del Habla (RAH), existen diversos paquetes software orientados a la construcción de sistemas de RAH de gran vocabulario, con habla continua e independientes de locutor. Entre ellos cabe destacar Sphinx [83, 190], HTK [225, 227], Julius [96], RASR [173] o Kaldi [143].

El presente capítulo describe las principales características de tres de los paquetes más utilizados: el Sphinx (de la Carnegie Mellon University) y el HTK (de la Universidad de Cambridge), así como Kaldi, un toolkit relativamente novedoso que, no obstante, ha conseguido hacerse rápidamente un lugar relevante dentro del ecosistema de entornos de desarrollo de sistemas de RAH.

El capítulo finaliza con la presentación de Sautrela [122, 123], un entorno de desarrollo flexible y modular para las Tecnologías del Habla, mostrando sus principales características y realizando una comparativa frente a los entornos de desarrollo previamente descritos.

2.2. CMU Sphinx

CMU Sphinx (`http://cmusphinx.org`) hace referencia a un conjunto de herramientas desarrolladas principalmente en la Universidad Carnegie Mellon [99]. Se trata de un entorno software de referencia en lo que respecta tanto a la investigación como al desarrollo de sistemas de reconocimiento del habla. El proyecto CMU Sphinx ha dedicado gran empeño al desarrollo de aplicaciones prácticas, por lo que da soporte a múltiples lenguas y casos de uso, y favorece el desarrollo comercial de sus productos.

En el año 2000, fue liberado el código fuente de los componentes de CMU Sphinx, que en la actualidad está compuesto por:

- **PocketSphinx**: Se trata de una librería ligera escrita en C y que contiene un motor de reconocimiento para ser integrado en aplicaciones. Al tratarse de una implementación ligera, puede ser integrada en entornos con capacidad de cómputo limitada, tales como arquitecturas ARM.
- **Sphinx4**: Un sistema de reconocimiento del habla adaptable, configurable y escrito en Java.
- **SphinxTrain**: Un conjunto de herramientas para el entrenamiento de modelos acústicos.
- **SphinxBase**: Conjunto de librerías compartidas por el resto de componentes (excepto Sphinx4).
- **cmudict**: Diccionario de pronunciación en inglés. Es considerado como el diccionario de pronunciación de Inglés Americano de referencia en el área de las tecnologías del habla. Contiene en torno a 134000 palabras y sus correspondientes transcripciones fonéticas en base al inventario fonético ARPAbet [191].
- **cmuclmtk**: CMU-Cambridge Statistical Language Modeling Toolkit. Se trata de un conjunto de herramientas UNIX para la creación y validación de modelos de lenguaje estadísticos [49]. Este conjunto de herramientas puede ser utilizado además para generar modelos de lenguaje compatibles con los decodificadores pocketsphinx y Sphinx4.

El sistema de reconocimiento CMU-Sphinx ha ido integrando las diferentes innovaciones que se daban en el área del reconocimiento automático del habla a lo largo del tiempo, dando lugar a sucesivas versiones:

Sphinx Versión inicial del sistema CMU Sphinx. Se trataba de un sistema de reconocimiento de habla continua, gran vocabulario e independiente de locutor, basado en HMMs discretos y modelos de lenguaje de n-gramas [99, 98, 100, 101].

Sphinx-2 Incluía el uso de HMMs semi-continuos y ofrecía un equilibrio entre rendimiento y velocidad ideal para ser integrado en aplicaciones en tiempo real. Contaba además con una licencia de tipo BSD, lo que facilitaba su comercialización [80, 81, 85].

Sphinx-3 Un sistema principalmente orientado a la investigación y basado en HMMs continuos. En evolución constante y unido a la herramienta de entrenamiento Sphinxtrain, ofrecía un conjunto moderno de técnicas de modelado y decodificación [142, 47, 82, 146].

Sphinx-4 Es un rediseño completo del motor de reconocimiento de Sphinx, basado en el lenguaje de programación Java y con el objetivo de crear un entorno flexible para la investigación en RAH. Sphinx-4 es producto de la colaboración entre el grupo Sphinx de la Carnegie Mellon University, Sun Microsystems Laboratories,

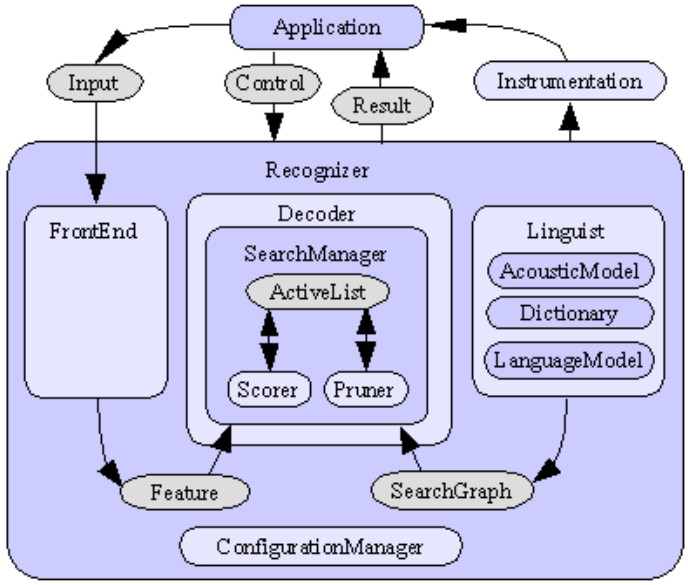


Figura 2.1 La arquitectura software del reconocedor Sphinx-4 está compuesta por tres bloques que acometen las tres tareas principales: la generación del grafo de búsqueda (*Linguist*), la obtención de vectores de parámetros a partir de la entrada de audio (*FrontEnd*) y la decodificación (*Decoder*). La mayor parte de los componentes representan interfaces de Java, lo que le confiere una gran flexibilidad.

Mitsubishi Electric Research Labs (MERL), y Hewlett Packard (HP), con contribuciones de la University of California en Santa Cruz (UCSC) y el Massachusetts Institute of Technology (MIT) [95, 190]. La Figura 2.1 muestra la arquitectura software del decodificador de Sphinx-4.

PocketSphinx Es la versión ligera del reconocedor Sphinx, heredera de Sphinx-2 pero con un rendimiento equiparable a Sphinx-3. Se trata de un sistema orientado al desarrollo de aplicaciones que requieran de una interfaz hablada, tanto en entornos de escritorio como móviles [83].

2.3. HTK

HTK (Hidden Markov Model Toolkit, `http://htk.eng.cam.ac.uk`) [225, 227] es un paquete de software creado para diseñar y manipular modelos ocultos de Markov (HMM). Se trata posiblemente del entorno de desarrollo de referencia en la comunidad de investigación, y más recientemente, en el mundo comercial. HTK fue desarrollado por Steven Young en el Grupo de Habla, Visión y Robótica del Departamento de Ingeniería de la Universidad de Cambridge (CUED) en 1989. En 1993 pasó a ser software comercial distribuido por Entropic, hasta que en 1999 la compañía fue adquirida por Microsoft, retornando posteriormente los derechos al CUED. A partir de septiembre de

2000, HTK es ofrecido como software de código abierto con la intención de convertirse en una plataforma que proporcione el conjunto de herramientas para el desarrollo de la investigación en Reconocimiento Automático del Habla (RAH). Desde su creación, HTK ha sido una plataforma de referencia para el desarrollo de las tecnologías del RAH, y sus sucesivas versiones [223, 221, 220, 75, 74, 92, 65, 91, 69] han reflejando la evolución de dichas tecnologías:

HTK 1.0 Versión inicial e interna del CUED. Contaba con un conjunto limitado de documentación, librerías y comandos. Permitía utilizar HMMs con mezclas de Gaussianas de covarianza diagonal o completa.

HTK 1.1-1.5 Primera versión de lanzamiento. El paquete contaba ya con un manual de referencia, y las sucesivas versiones fueron añadiendo nuevas características al entorno de desarrollo: caché en las distribuciones de probabilidad, métodos de poda en la decodificación y la re-estimación, ligaduras en los parámetros de los HMMs, flujos de datos de entrada múltiples, comando de edición de modelos, alineamiento forzado, compresión en los ficheros de parámetros y normalización cepstral, entre otras muchas.

HTK 2.0-2.2 En esta versión se rediseñó una gran parte de las librerías y comandos, y la documentación pasó a formar parte del HTK Book. Se integraban diversas tecnologías, entre las que podemos destacar: HMMs con distribuciones de probabilidad discretas, el uso de trifonemas inter-palabra, la obtención de lattices y decodificación N-best, la estimación de contextos fonéticos mediante árboles de decisión, la entrada de audio en tiempo real, los ficheros de configuración, la detección de silencio basada en energía, los modelos de pronunciación probabilísticos y la adaptación al locutor mediante MLLR y MAP.

HTK 3.0-3.4 Añadió compatibilidad con el lenguaje de programación C++, e incluyó una serie de tecnologías entre las que podemos destacar: la extracción de parámetros Perceptual Linear Prediction (PLP), la normalización de la longitud del tracto vocal, la normalización cepstral basada en clústers, una herramienta de modelado de lenguaje, una herramienta de post-proceso de lattices, la estimación de transformaciones lineales en el espacio de los parámetros, la adaptación al locutor mediante MLLR y CMLLR, el entrenamiento discriminativo de HMMs y un decodificador de gran vocabulario con modelos de lenguaje de trigramas.

HTK 3.5 Recientemente ha sido anunciado el lanzamiento de una nueva versión, que incluye el modelado acústico basado en redes neuronales profundas (Deep Neural Networks o DNNs), soporte para el cómputo basado en GPUs y modelos de lenguaje basados en redes neuronales recurrentes (Recurrent Neural Networks o RNNs) [219, 240].

2.3.1. Arquitectura

HTK se compone de un conjunto de herramientas o comandos y módulos de librerías en C/C++, todo ello ofrecido como proyecto de código abierto. A pesar de tratarse de un entorno de desarrollo para el entrenamiento y la evaluación de HMMs, elementos

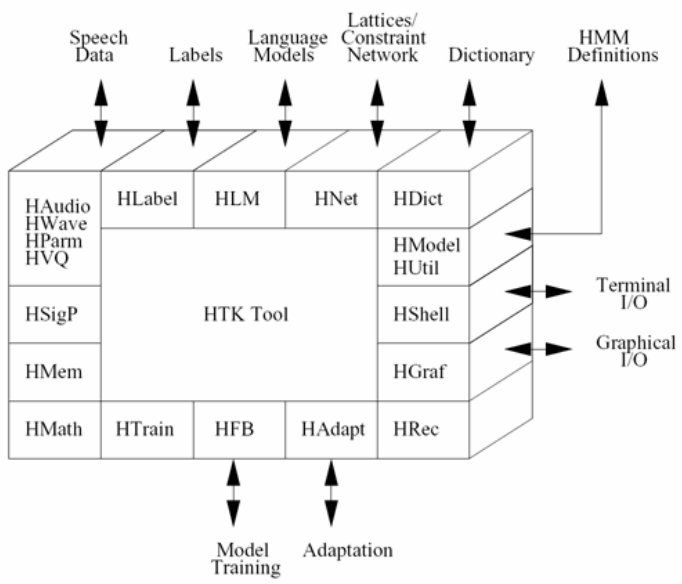


Figura 2.2 Aquitectura software de HTK. La mayor parte de la funcionalidad del entorno de desarrollo está contenida en el conjunto de bibliotecas, cada una de ellas destinada a una tarea específica.

de uso genérico, su diseño está claramente orientado a la creación de aplicaciones de procesamiento del habla basadas en HMMs, en particular, sistemas de reconocimiento del habla. Así, la mayor parte de su infraestructura tiene por objeto una de las dos tareas principales: en primer lugar, estimar los parámetros de un conjunto de HMMs dado un conjunto de sentencias transcritas, y en segundo lugar, obtener la transcripción de una señal de voz.

La Figura 2.2 muestra el conjunto de bibliotecas que componen la arquitectura de software de HTK [227]. Este conjunto de librerías está compuesto por módulos encargados de cuestiones específicas y que contienen la mayor parte de la funcionalidad del entorno de desarrollo. Así, el módulo HMem es el encargado de la gestión de memoria, el módulo HMath ofrece soporte matemático, el módulo HDict proporciona una interfaz con los diccionarios, el módulo HWave da soporte a los ficheros de audio mientras que el módulo HParam da soporte a los ficheros de parámetros, etc.

Las herramientas o comandos de HTK hacen uso del conjunto de librerías y ofrecen una interfaz de línea de comando al usuario final. Los comandos pueden ser agrupados en función de la fase de procesamiento en la que son utilizados:

Preparación de datos. La adecuación de una base de datos para entrenar HMMs implica la adaptación de los ficheros de datos y sus correspondientes transcripciones. HTK contiene un conjunto de comandos que permiten adaptar a su formato propio tanto los ficheros de audio como sus transcripciones. A pesar de soportar la parametrización "al vuelo" de ficheros de audio, resulta más eficiente realizar una única parametrización inicial de estos ficheros y generar una base de datos

ya parametrizada. Por otro lado, también contiene comandos cuya finalidad es el análisis y la monitorización del conjunto de datos generado.

Entrenamiento de modelos. Esta fase de procesamiento parte de la definición inicial de las topologías de los HMMs a entrenar. Una vez definidas, existen diferentes estrategias de entrenamiento en función de si se cuenta o no con un subconjunto de datos con segmentación fonética. HTK cuenta con diversos comandos que permiten adaptar el entrenamiento de los modelos a diversas situaciones: inicialización de modelos en base a una segmentación fonética, inicialización de componentes Gaussianas en base a una media y varianza global, y re-estimación de HMMs en base a una transcripción fonética. Cuenta también con herramientas de adaptación de modelos a locutores específicos. Además, para evitar el problema de la insuficiencia de datos al aumentar la complejidad de los modelos, HTK ofrece la posibilidad de ligar (es decir, compartir) los parámetros de los modelos.

Reconocimiento. HTK ofrece dos comandos que desarrollan el proceso de decodificación o reconocimiento: HVite y HDecode. El primero es una herramienta genérica de reconocimiento que toma una red que describe las posibles secuencias de palabras a reconocer, un diccionario de pronunciación y un conjunto de HMMs, y obtiene la secuencia de estados de HMMs más probable para cada secuencia de datos a reconocer. La red de palabras puede representar un modelo de lenguaje o una gramática regular específica de una tarea. El comando HVite también puede generar redes de fonemas, re-evaluar redes de fonemas y realizar alineamientos forzados. HDecode, por el contrario, es un decodificador orientado a tareas de gran vocabulario. A diferencia del primer comando, su optimización conlleva una serie de restricciones de uso, tales como la limitación a modelos de lenguajes de tri-gramas, el modelado de trifonemas inter-palabra o la inclusión automática de silencios y pausas cortas entre palabras. HTK cuenta además con una herramienta genérica de re-estimación de redes de fonemas o palabras que permite aplicar en una segunda fase de reconocimiento modelos de lenguaje más complejos.

Análisis. El rendimiento de un reconocedor es evaluado comparando la transcripción original de un subconjunto de test frente a la salida del decodificador. HTK cuenta con una herramienta de análisis numérico y gráfico de resultados con este fin.

2.3.2. ATK: An Application Toolkit for HTK

ATK [226] es una API en C++ que recubre el conjunto de librerías de HTK y que ha sido diseñada para facilitar la construcción de aplicaciones que hagan uso de las librerías de HTK. Se trata de una arquitectura multihilo y modular (basada en componentes), que además incluye Flite [25], un sencillo sintetizador de voz en inglés desarrollado por la CMU. ATK es un entorno diseñado especialmente para la creación de aplicaciones de diálogo. El diagrama de dependencias de la Figura 2.3 muestra la estructura de recubrimiento que da lugar a la interfaz de aplicación y que ofrece un nivel de abstracción más adecuado para la creación de aplicaciones.

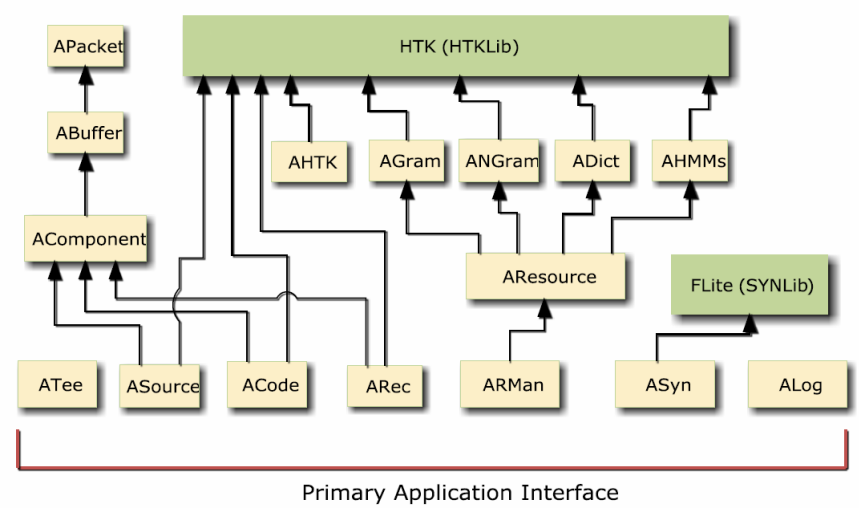


Figura 2.3 La aquitectura software de ATK ofrece una API de recubrimiento sobre el conjunto de librerías de HTK que facilita el desarrollo de aplicaciones, y en especial aquellas que requieran de un módulo de diálogo.

2.4. Kaldi

Kaldi (`http://kaldi-asr.org`) es un entorno de desarrollo escrito en C++ y orientado a la investigación y desarrollo de sistemas de reconocimiento y clasificación en la área de las tecnologías del habla [143]. Sus orígenes se remontan a un workshop del año 2009 en la Johns Hopkins University, donde se desarrolló cierto código orientado a la implementación de la técnica denominada Subspace Gaussian Mixture Model (SGMM). El código estaba a su vez basado en HTK, pero como producto de un workshop celebrado en la Brno University of Technology al año siguiente, se llevó a cabo un rediseño de la arquitectura de software, desechando dicha dependencia. Actualmente, Kaldi es un entorno de desarrollo flexible y extensible con las siguientes características:

- **Librería de álgebra lineal**. Kaldi contiene una librería matricial que recubre las rutinas estándares BLAS y LAPACK.
- **Mecanismo genérico de entrada/salida**. Kaldi cuenta con un mecanismo flexible de entrada/salida que permite representar infinidad de diferentes configuraciones de entrada/salida en cualquiera de los comandos de la herramienta.
- **Conjunto extenso de comandos sencillos**. Frente a otros entornos de desarrollo, que cuentan con un conjunto reducido de comandos que realizan infinidad de tareas diferentes, Kaldi está compuesto por multitud de herramientas, cada una de ellas con una única finalidad.
- **Arquitectura de decodificación flexible y simplificada**. Un decodificador está compuesto, por un lado, por distribuciones de probabilidad que representan

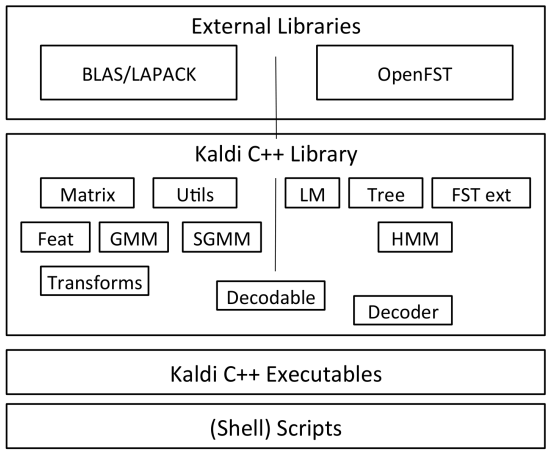


Figura 2.4 Aquitectura software de Kaldi. El entorno de desarrollo tiene dos dependencias externas: por un lado, las librerías de álgebra lineal BLAS y LAPACK, y, por otro, la librería OpenFST, que da soporte a los transductores de estados finitos. Los módulos que conforman la librería de Kaldi pueden dividirse en dos subconjuntos: los relacionados con el modelado acústico (únicamente en lo que se refiere a las distribuciones de probabilidad) y que únicamente dependen de las librerías numéricas, y el resto de módulos (topología de HMMs, modelos de contextos acústicos, modelos léxicos, modelos de lenguaje y el decodificador), que hacen uso de la librería se transductores. Una única clase (*Decodable*) sirve de unión entre ambos conjuntos de módulos. La librería de Kaldi es accedida a través de un extenso conjunto de comandos o herramientas escritas en C++, las cuales son a su vez instanciadas y combinadas desde scripts de Shell de Unix/Linux.

el modelado acústico y, por otro lado, por conjuntos de transductores que pueden representar la topología de los modelos acústicos, los modelos léxicos y el modelo de lenguaje. cabe destacar las siguientes características:

- La separación entre las distribuciones de probabilidad y la topología de los modelos fonéticos permite implementar modelos de distribuciones de probabilidad alternativos a las mezclas de Gaussianas, como por ejemplo las redes neuronales.
- Los componentes restantes son representados mediante transductores de estados finitos (FST, por sus siglas en inglés), ofreciendo operaciones genéricas tales como como la composición, la minimización y la determinización, todo ello en base a la librería OpenFst (`http://www.openfst.org`).

- **Conjunto de recetas**. Kaldi cuenta con un conjunto de *recetas* o ejemplos que muestran cómo construir desde cero un sistema de reconocimiento del habla a partir de bases de datos disponibles para todo el mundo, tales como las distribuidas por el Linguistic Data Consortium (LDC).

En la actualidad, Kaldi es un conjunto de herramientas en continuo desarrollo que integra la mayor parte de las tecnologías de vanguardia en reconocimiento automático

del habla. Permite realizar multitud de transformaciones lineales de parámetros de entrada y adaptación al locutor, el entrenamiento discriminativo de modelos acústicos basados en GMMs, la estimación y uso de modelos acústicos basados en Deep Neural Networks (DNNs), el cómputo sobre GPUs, etc.

2.5. Sautrela

Sautrela[1] (http://sautrela.org) es un entorno de desarrollo flexible y modular para las Tecnologías del Habla escrito en el lenguaje de programación Java™ y creado en el Grupo de Trabajo en Tecnologías Software (GTTS, http://gtts.ehu.es) del Departamento de Electricidad y Electrónica de la Facultad de Ciencia y Tecnología de la Universidad del País Vasco/Euskal herriko Unibertsitatea (UPV/EHU) [122, 123].

Los orígenes de Sautrela se remontan a los orígenes mismos del grupo GTTS, cuando se planteó la necesidad de desarrollar un sistema de RAH propio. En un inicio, dicha tarea fue acometida en base al lenguaje de programación C, pero debido a la complejidad del proyecto, la migración a un lenguaje orientado a objetos resultó casi natural. El lenguaje de programación adoptado fue Java, que a pesar de no ofrecer por aquellos tiempos un rendimiento comparable a los lenguajes compilados como el C o C++, sí que ofrecía un entorno de programación más atractivo (una orientación a objetos más pura y la extensa API de Java), así como progresivas mejoras en la tecnología propia de la Máquina Virtual (el rendimiento de las aplicaciones Java mejoraba con cada una de las revisiones de la misma).

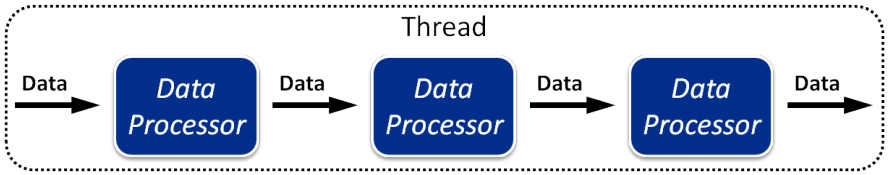
Este desarrollo coincidió en tiempo y forma (lenguaje de programación Java) con el proyecto Sphinx-4, cuyo *front-end* o módulo de extracción de características (parametrización) adoptó el esquema de abstracción de procesamiento que se muestra en la Figura 2.5. La parametrización no era sino el producto de una serie de procesadores o filtros que transformaban la señal o datos de entrada. El uso de interfaces a la hora de formalizar tanto los procesadores como los datos de entrada/salida confería al *front-end* de Sphinx-4 la cualidad de ser fácilmente extensible: todo componente que implementase la interfaz de procesamiento podía formar parte de la cadena de procesamiento que daba lugar a la extracción de características.

Sautrela adoptó dicho mecanismo no solo en lo que respecta a la extracción de características, sino como la naturaleza misma del entorno de desarrollo, pasando de la pretensión inicial de ser un sistema de RAH a convertirse en un sistema de procesamiento de la información de propósito general. Si bien la mayoría de sus componentes (*Procesadores*) han sido diseñados en torno al desarrollo de sistemas de RAH, su diseño tiene por objeto la creación de un conjunto versátil, modular y extensible de componentes que permita abordar otras muchas tareas relacionadas con las Tecnologías del Habla. Las siguientes características resumen la naturaleza de Sautrela:

- **Software Libre.** Sautrela está licenciado mediante BSD (Licencia BSD simplificada o licencia FreeBSD), lo que permite su libre redistribución y modificación.

[1] Sautrela es el título del último poema del primer libro impreso en Euskera, *Linguae Vasconum Primitiae* ("Primicias de la lengua de los vascones"), escrito por Bernat Etxepare e impreso en Burdeos en el año 1545. El término Sautrela podría referirse a una danza de procedencia italiana, en concreto de Venecia (*saltarella*).

a)



b)

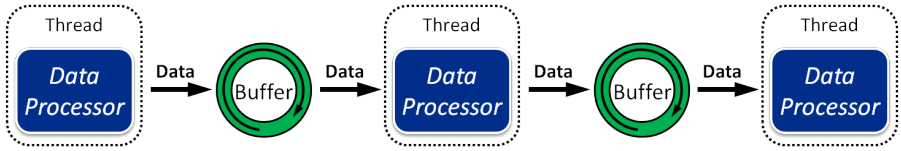


Figura 2.5 Comparativa entre el esquema de ejecución del *front-end* de Sphinx-4 y Sautrela. El Frontend *front-end* de Sphinx-4 (a) es ejecutado por un único hilo, generando una cadena de procesamiento que es activada bajo demanda de datos del último procesador. Una *Engine* o cadena de ejecución de Sautrela (b) ejecuta cada uno de los procesadores de manera concurrente, interconectando dichos procesos mediante buffers intermedios.

- **Fácil de instalar.** Sautrela solo requiere de la instalación previa de Java. Si ya se cuenta con Java, no se requiere de ninguna instalación o compilación adicional: basta con descargar el fichero `Sautrela.jar`.
- **Portable.** Al estar escrito únicamente en Java, Sautrela es un entorno portable que puede ser ejecutado en multitud de plataformas sin requerir compilación alguna.
- **Entorno de propósito general.** Como ya se ha comentado previamente, Sautrela cuenta con un conjunto de componentes que permiten la creación de sistemas de RAH. No obstante, su diseño modular hace posible el desarrollar multitud de sistemas de naturaleza distinta a la de un sistema de RAH.
- **Modular.** Sautrela se basa en el modelo de componentes software JavaBeans™ [10], integrando de forma natural tanto componentes propios como componentes desarrollados por terceros.
- **Flexible.** El extenso uso de las interfaces le confiere gran capacidad de adaptación a diferentes casos de uso.
- **Concurrente.** A diferencia del *front-end* de Sphinx-4, que utiliza un único hilo de ejecución, los procesadores de Sautrela son ejecutados de manera concurrente (véase Figura 2.5), ajustándose perfectamente a las arquitecturas hardware multicore actuales y permitiendo aprovechar de una manera natural los recursos del sistema.

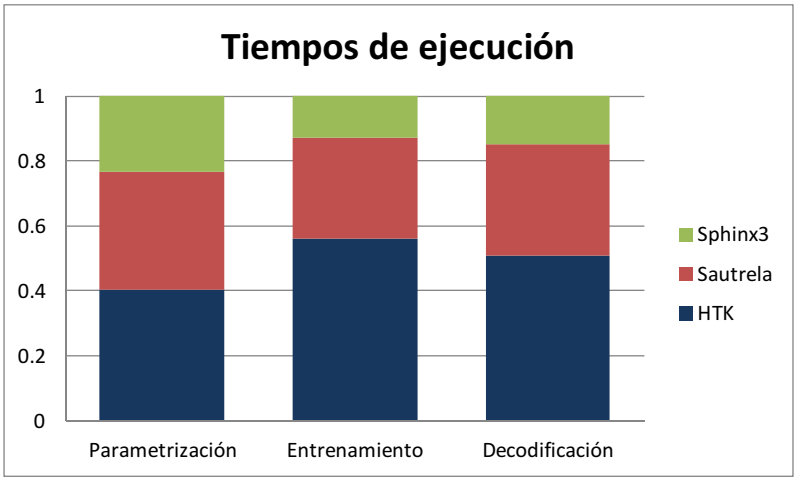


Figura 2.6 Comparativa de los tiempos de ejecución para las fases de parametrización, entrenamiento y decodificación acústico-fonética.

- **Orientado a la investigación.** En su diseño siempre ha prevalecido la flexibilidad frente al rendimiento computacional. Sautrela no es una plataforma diseñada para construir sistemas de RAH comerciales, sino para estudiar y desarrollar nuevas tecnologías que puedan dar lugar a mejoras de rendimiento. Ello no significa que el procesamiento vaya a ser más lento que en otras plataformas.Únicamente ha de constatarse que no ha sido diseñado con ese fin. La Figura 2.6 muestra una comparativa de los tiempos requeridos para la extracción de características, el entrenamiento de modelos acústicos y la decodificación acústico-fonética.

Sautrela tiene por finalidad la creación de cadenas de procesamiento de la información, denominadas *Engine*s. Este procesamiento es llevado a cabo por un conjunto flexible y extensible de *Procesadores* que transforman los flujos de información compuestos por secuencias de datos, cuya naturaleza puede ser de texto o numérica, escalar o vectorial. En base al modelo de componentes software JavaBeans™, Sautrela permite la integración de *plugins* que contengan *Procesadores* y *Comandos* desarrollados por terceros, y define además un mecanismo para la generación automática de documentación de las *Engines*, los *Comandos* y los *Procesadores*.

A diferencia de otros entornos de desarrollo, más focalizados en las tareas de entrenamiento y decodificación, para Sautrela, tanto la re-estimación de modelos como la decodificación, son un elemento más de una cadena de procesamiento. En cierta manera, esta característica se asemeja al método de entrada/salida de Kaldi, donde los comandos cuentan con un mecanismo versátil que permite la concatenación de subprocesos.

Sautrela define un sencillo conjunto de interfaces que ofrecen una abstracción funcional de los modelos basados en autómatas de estados finitos ponderados (WFSA, por sus siglas en inglés). En base a la abstracción funcional ofrecida por los interfaces, Sautrela formaliza un mecanismo unificado de decodificación que, dado un WFSA

y una secuencia de observaciones, obtiene la secuencia más probable de transiciones. Sautrela cuenta ya con un conjunto flexible de modelos que implementa la interfaz WFSA, y dicho conjunto es fácilmente extensible: todo modelo que implemente la interfaz WFSA podrá ser integrado en Sautrela. De manera análoga a la decodificación, Sautrela formaliza un mecanismo unificado de re-estimación para todo aquel modelo que implemente la interfaz WFSA.

Por último, Sautrela implementa el denominado modelo de Markov por capas (Layered Markov Model, o LMM) [121], que integra en un único WFSA distintos niveles de conocimiento, representados por conjuntos de WFSAs. Cada capa, o nivel de conocimiento, modela sus clases en función de las unidades de la capa inferior, de manera que existe una correspondencia directa entre las observaciones de los modelos de una capa y los modelos de la capa inferior (en un sistema de RAH, por ejemplo, las capas de un LMM corresponderían con los conjuntos de modelos acústicos, léxicos y de lenguaje). Al implementar la interfaz WFSA, un LMM puede ser utilizado tanto para construir sistemas de reconocimiento de secuencias de patrones como para guiar el proceso de re-estimación de los modelos: el formalismo de los LMM ofrece una excepcional flexibilidad a la hora de diseñar sistemas de reconocimiento o decodificación, y de aprendizaje o re-estimación.

Capítulo 3

Sautrela: Arquitectura de Software

La mayoría de las tareas que se abordan dentro de las tecnologías del habla pueden verse como complejos sistemas de procesamiento de la información: un reconocedor del habla no es sino un decodificador que trata de extraer un mensaje que ha sido previamente codificado en una señal sonora, y un verificador de la lengua trata de comprobar si una lengua ha sido o no utilizada en la codificación acústica de un mensaje. Este tipo de sistemas se componen normalmente de fases independientes de procesamiento (eliminación de eco y reverberación, extracción de información espectral, reducción de ruido, decorrelación, compensación de canal, modelado acústico, etc.) cuyo encadenamiento da lugar al procesamiento global. Sautrela [122, 123] se basa en la abstracción de componentes o agentes de procesamiento (`DataProcessor`) que al encadenarse dan lugar a sistemas complejos de procesamiento (`Engine`). La herramienta Sautrela define una interfaz genérica de datos (`Data`) que encapsula tanto los datos a procesar como las señales de control. Ambos fluyen a través de una máquina o sistema complejo (`Engine`), compuesta por agentes independientes de procesamiento (`DataProcessor`) que son interconectados mediante registros de almacenamiento intermedio (`Buffer`).

3.1. Datos y señales de control

Una engine tiene por finalidad el procesamiento de secuencias de datos. Las estructuras de flujo vienen definidas mediante señales de control. Una secuencia consta de una cabecera (`StreamBegin`), una secuencia de datos alfanuméricos (`IntData`, `DoubleData` o `StringData`) y una señal de finalización de flujo (`StreamEnd`). Existe a su vez una señal (EOS) que notifica la no existencia de más secuencias de datos a procesar. Todos ellos, señales y datos alfanuméricos, implementan la interfaz genérica de datos (`Data`). A continuación se describe el conjunto de datos y señales de control definidos por Sautrela:

- **Data:** Interfaz genérica de datos que aglutina los datos a procesar y las señales de control.

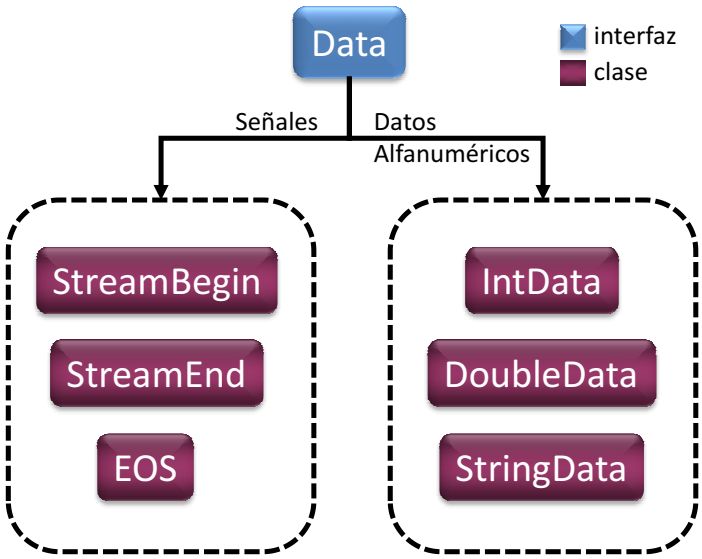


Figura 3.1 Señales y datos alfanuméricos, ambos implementan la interfaz genérica de datos.

- **IntData:** Vector de enteros.
- **DoubleData:** Vector de reales.
- **StringData:** Cadena de caracteres.
- **StreamBegin:** Marcador de inicio de secuencia de datos. Puede contener cualquier tipo de información adicional sobre el recurso en cuestión (el formato de audio, su transcripción fonética, el identificador del locutor, etc).
- **StreamEnd:** Marcador de final de secuencia de datos.
- **EOS:** Marcador de final de procesamiento (*End Of Stream*). Informa de la no existencia de más secuencias de datos a procesar. Los procesadores deben finalizar su tarea una vez hayan recibido esta señal.

Los datos numéricos (`IntData` y `DoubleData`), pueden representar tanto una secuencia temporal de datos unidimensionales (flujo de información escalar), como un valor multidimensional instantáneo (flujo de información vectorial). Por ejemplo, un sencillo procesador de adquisición de audio podría devolver bloques de audio mediante vectores de enteros, en cuyo caso el vector representaría una secuencia temporal. Por el contrario, otro procesador de adquisición de audio que recibiese la información de un array de micrófonos podría generar vectores de enteros que representarían una información vectorial instantánea. Dado que la interpretación escalar (secuencial) o vectorial (instantánea) de la información generada depende de cada procesador, es el usuario de Sautrela quien debe tenerlo en cuenta al diseñar una engine compuesta por distintos procesadores.

Listado 3.1 Declaración de la interfaz `DataProcessor`. El procesador debe leer datos del buffer de entrada y volcarlos al buffer de salida.

```
public interface DataProcessor {
   public void process(Buffer in, Buffer out)
      throws DataProcessorException;
}
```

Listado 3.2 Ejemplo de implementación de la interfaz `DataProcessor`. Un procesador consume los datos de su buffer de entrada, los procesa y los vuelca en su buffer de salida. En cuanto recibe una señal `EOS`, finaliza el procesamiento volcando la señal `EOS` en el buffer de salida.

```
package path.to.mypackage;

public class MyProcessor Implements DataProcessor {
   public void process(Buffer in, Buffer out)
      throws DataProcessorException {
      Data d = null;
      while ((d = in.read()) != Data.EOS){
         ...
      }
      ...
      out.write(d);
   }
}
```

3.2. Procesadores

Los procesadores son unidades independientes de procesamiento que implementan una sencilla interfaz que les permite ser integrados en una engine. Dicha interfaz consta de un único método, que es el responsable de leer datos del buffer de entrada, procesarlos y volcar el resultado al buffer de salida. El procesamiento (el método implementado) debe concluir al recibir una señal de terminación (`EOS`), debiendo escribir en el buffer de salida todos los datos pendientes y la propia señal `EOS`, lo que encadenará la finalización de los procesadores posteriores. A su vez, la señal `EOS` debe ser generada por el primero de los módulos en la cadena de procesamiento tan pronto finalice la producción de los flujos de datos. En el Listado 3.1 puede verse la declaración de la interfaz (en lenguaje Java) de los procesadores, que está compuesta por un único método. El Listado 3.2 muestra un ejemplo de implementación de la interfaz.

Los procesadores son libres de transformar los flujos de datos, pudiendo generar más de un dato a partir de uno de entrada, filtrar datos, generar más de una secuencia de datos a partir de una secuencia de entrada, filtrar secuencias de datos completas, convertir secuencias escalares en vectoriales, vectoriales en escalares, modificar el tipo de información procesada, etc. En función de la tarea que realizan, los procesadores pueden ser clasificados en:

Generadores Se trata de procesadores que generan, cargan o adquieren datos y los vuelcan al buffer de salida (ignoran el buffer de entrada). Deben colocarse al inicio de la cadena de procesamiento ya que son, en esencia, la fuente de datos.

Transformadores Son aquellos procesadores cuya finalidad última es transformar los datos de entrada y volcar el resultado al buffer de salida. Pueden modificar parcial o completamente las secuencias de datos de entrada, filtrarlas o añadirles información.

Analizadores Su finalidad es analizar la información contenida en los datos de entrada, sin intención alguna de transformarlos. Como norma general, los analizadores deben volcar al buffer de salida el flujo completo de datos que reciben a la entrada. El resultado de su análisis puede ser volcado a un fichero de datos, a la salida estándar o ser mostrado gráficamente.

El Capítulo 4 y el Apéndice C muestran y describen el conjunto de procesadores incluidos en Sautrela. A continuación se discute más en profundidad el patrón de diseño para la creación e integración de nuevos procesadores en Sautrela.

3.2.1. Integración de procesadores: plugins

Un *plugin* es un conjunto de procesadores diseñados para algún tipo de aplicación específica. Basándose en el modelo de componentes software JavaBeans™ [10], Sautrela integra de forma natural tanto plugins propios como plugins desarrollados por terceros. En este sentido, la *introspección* [13] es una tecnología clave, ya que gracias a ella es posible instanciar y configurar los procesadores contenidos en un plugin y obtener la documentación asociada a cada procesador.

Se denomina introspección a la habilidad de examinar en tiempo de ejecución el tipo, campos, métodos y constructores de un objeto. Mediante la introspección es posible, por ejemplo, instanciar un objeto de una clase de la cual solo conocemos el nombre. Dicho nombre puede ser comunicado en tiempo de ejecución, pudiendo tratarse de una clase creada con posterioridad al código de instanciación. De igual manera, es posible descubrir sus campos y sus métodos, instanciar el objeto e interaccionar con él. El paquete `java.lang.reflect` y la clase `java.lang.Class` conforman la denominada *API de Reflexión* [13] que ofrece herramientas de introspección en Java.

La introspección suele utilizarse también para manipular objetos de clases desconocidas, pero que implementan algún patrón de diseño conocido. En Sautrela, el instanciador de engines no conoce de antemano ninguno de los procesadores, ya que tanto los plugins embebidos en Sautrela como los desarrollados por terceros son descubiertos durante el proceso de instanciación. El nombre de clase del procesador identifica de manera unívoca la clase del objeto a instanciar. Además, todo procesador debe implementar el patrón de diseño JavaBeans™, según el cual debe existir un constructor sin argumentos (que será utilizado para instanciar el procesador), y todas las propiedades públicas de la clase (a nuestros efectos, los parámetros del procesador) deben ser accesibles mediante métodos `get/set`: si una clase tiene una propiedad `color`, debe contar con un método `getColor` para acceder a dicho valor, y si se trata de una propiedad modificable, deberá también contar con el correspondiente método `setColor`. El patrón JavaBeans™ establece también un procedimiento para publicar información

relativa a un *bean*, que es como se denomina a los componentes que implementan el patrón de diseño (el procedimiento se describe brevemente en la Sección 3.2.2). La clase `java.beans.Introspector` contiene un conjunto de utilidades que se ocupan de las acciones de introspección de bajo nivel, y ofrece una API de reflexión de más alto nivel, específica y, en definitiva, más sencilla.

Basándose en el patrón de diseño JavaBeans, Sautrela es capaz de generar la documentación de un procesador, mostrando la descripción del procesador y cada uno de sus parámetros, así como sus tipos y valores por defecto. Nótese que Sautrela desconoce de antemano la existencia del procesador. Partiendo del nombre completo de la clase (que incluye el nombre de paquete, lo que se conoce también como nombre de clase totalmente cualificado), los mecanismos de introspección permiten detectar los parámetros y extraer toda la documentación asociada. La siguiente línea de comando[1], por ejemplo, mostrará la documentación del procesador `MyProcessor` contenido en el paquete `path.to.mypackage` :

```
$ sautrela -help path.to.mypackage.MyProcessor

PROCESSOR: MyProcessor (path.to.mypackage.MyProcessor)

  This is the documentation of MyProcessor

PARAMETERS:

  myParam1 [int,5] - description of myParam1
  myParam2 [String,"something"] - description of myParam2
  myParam3 [double,1.0] - description of myParam3
  myParam4 [int,10] - description of myParam4
```

De no existir dos procesadores con el mismo nombre sencillo de clase (`MyProcessor` en el caso anterior), es posible utilizar dicho nombre sencillo en vez del nombre de clase totalmente cualificado, dado que no existe ambigüedad[2]:

```
$ sautrela -help MyProcessor

PROCESSOR: MyProcessor (path.to.mypackage.MyProcessor)

  This is the documentation of MyProcessor

PARAMETERS:

  myParam1 [int,5] - description of myParam1
  myParam2 [String,"something"] - description of myParam2
  myParam3 [double,1.0] - description of myParam3
  myParam4 [int,10] - description of myParam4
```

[1] El comando `sautrela` es en realidad un alias de: `<path_to_java> -jar <path_to_Sautrela.jar>`

[2] Al inicializarse, Sautrela escanea los plugins en busca de componentes. En caso de encontrar anbigüedad con algún nombre sencillo de clase, notificará que dicho nombre no podrá ser utilizado. Todos los componentes integrados en Sautrela tienen nombres no ambiguos, pero ello no impide que un plugin de terceros pueda incluir un componente que genere ambigüedad frente a los ya existentes.

3.2.2. Creación de nuevos plugins

Como se ha explicado previamente, un plugin está formado por un conjunto de clases que deben implementar la interfaz `DataProcessor` y deben cumplir además con el patrón de diseño JavaBeans™. Para facilitar la creación de nuevos procesadores, Sautrela incorpora una clase, `AbstractProcessor`, que implementa la mayor parte de la funcionalidad impuesta por el patrón de diseño JavaBeans™[3] y delega en las clases que la extiendan la parte más sencilla y específica de cada procesador: el procesamiento, la documentación y los métodos de lectura/escritura de las propiedades. El Listado 3.3 muestra la implementación del procesador `MyProcessor` a partir la extensión de la clase `AbstractProcessor`. Pueden diferenciarse tres secciones de código:

- El método `process` realiza el procesamiento de los datos de entrada, volcando el resultado al buffer de salida.
- El método `editBeanInfo` es el encargado de suministrar toda la documentación relativa al procesador: la descripción del procesador y cada uno de los parámetros.
- Los métodos `get/set` definen el conjunto de propiedades del procesador y el acceso a las mismas. Sautrela solo mostrará los parámetros modificables, descartando aquellos que no cuenten con un método `set`.

Estamos ahora en condiciones de aclarar la procedencia de la información que conforma la documentación de los procesadores. Las descripciones son suministradas por el método `editBeanInfo`. Los tipos de los parámetros son obtenidos a través de la introspección (consultando los tipos de datos asociados a los métodos `get/set`). Finalmente, los valores por defecto se obtienen instanciando un procesador y consultando los valores de sus propiedades.

Por último, el patrón JavaBeans exige que el fichero MANIFEST.MF (incluido en las aplicaciones Java) contenga una declaración expresa referida a la clase en cuestión, a la cual se añade también otra declaración exigida por Sautrela. En resumen, el manifiesto debe contener:

```
Name: path.to.mypackage.MyProcessor.class
Java-Bean: True
Sautrela-Processor: True
```

Una vez empaquetado el plugin en un fichero JAR, los nuevos procesadores están listos para ser integrados en una engine. A continuación se muestran las fases de rastreo que realiza Sautrela en busca de plugins propios y de terceros:

1. Se cargan todos los procesadores contenidos en el fichero `Sautrela.jar`.
2. Se rastrea el directorio donde reside el fichero `Sautrela.jar` en busca de otros ficheros JAR que pudieran contener procesadores.
3. Se rastrea el directorio desde donde se ha ejecutado Sautrela en busca de otros ficheros JAR que pudieran contener procesadores.

[3] Si bien el patrón de diseño JavaBeans™ es sencillo (se limita a la existencia del constructor sin argumentos y a los métodos `get/set` para acceder a las propiedades), el procedimiento de publicación de información resulta sustancialmente más complejo.

Listado 3.3 Ejemplo de implementación de un procesador extendiendo `AbstractProcessor`. Toda propiedad de un *Bean* debe contar con un método de lectura `get`, y aquellas propiedades que sean modificables contarán también con un método de escritura `set`. El método `editBeanInfo` suministra toda la documentación del procesador.

```
package path.to.mypackage;

import edu.gtts.sautrela.engine.AbstractProcessor;
import edu.gtts.sautrela.engine.Buffer;
import edu.gtts.sautrela.engine.DataProcessorException;
import java.beans.BeanInfo;
import java.beans.PropertyDescriptor;

public class MyProcessor extends AbstractProcessor {

   public void process(Buffer in, Buffer out)
      throws DataProcessorException {
      Data d = null;
      while ((d = in.read()) != Data.EOS){ ... }
      out.write(d);
   }

   public void editBeanInfo(BeanInfo info) {
      info.getBeanDescriptor().setShortDescription(
         "This is the documentation of MyProcessor"
      );
      for(PropertyDescriptor pd : info.getPropertyDescriptors()){
         if (pd.getName().equals("myParam1")) {
            pd.setShortDescription("description of myParam1");
         } else if (pd.getName().equals("myParam2")) {
            pd.setShortDescription("description of myParam2");
         } else if (pd.getName().equals("myParam3")) {
            pd.setShortDescription("description of myParam3");
         } else if (pd.getName().equals("myParam4")) {
            pd.setShortDescription("description of myParam4");
         }
      }
   }

   public int getParamName1(){ ... }
   public void setParamName1(int newValue){ ... }

   public String getParamName2(){ ... }
   public void setParamName2(String newValue){ ... }

   public double getParamName3(){ ... }
   public void setParamName3(double newValue){ ... }

   public int getParamName4(){ ... }
   public void setParamName4(int newValue){ ... }
}
```

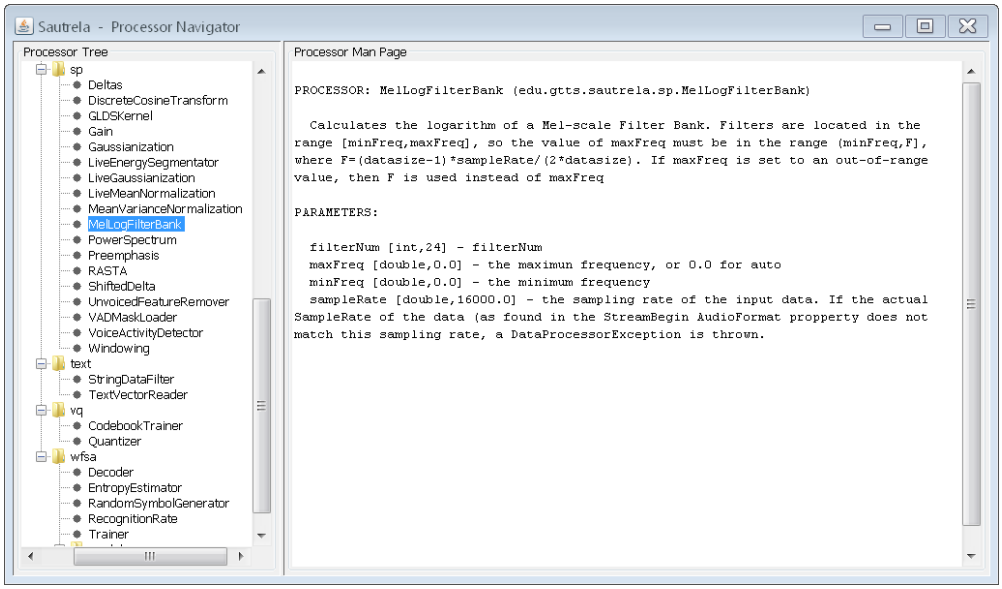


Figura 3.2 La aplicación gráfica *Processor Navigator* muestra el conjunto de procesadores contenidos en plugins propios y de terceros. Cada procesador va acompañado de su correspondiente documentación.

Por tanto, para poder hacer uso de plugins de terceros es suficiente con depositar el correspondiente fichero JAR bien junto al fichero `Sautrela.jar`, bien en el directorio desde donde estemos ejecutando Sautrela. Si bien es posible acceder a la documentación de cualquier procesador (propio o de terceros) mediante la línea de comando mostrada anteriormente, ello requiere conocer el nombre de la clase. Sautrela incluye una aplicación gráfica, *Processor Navigator*, que permite consultar el conjunto de procesadores contenidos en plugins propios y de terceros, así como su correspondiente documentación. La aplicación, que se muestra en la Figura 3.2, puede ser ejecutada mediante la línea de comando:

```
$ sautrela ProcessorNavigator
```

3.3. Buffers

Los buffers son estructuras FIFO (*First In First Out*) que permiten interconectar los procesadores que, en Sautrela, se ejecutarán en paralelo. Un procesador, el productor, introducirá elementos en el buffer, mientras que otro, el consumidor, los extraerá. Los buffers permiten la interconexión eficiente tanto de procesadores con velocidades de transferencia diferente, como de procesadores con fluctuaciones de velocidad. Sautrela permite al usuario configurar los buffers mediante los dos parámetros que se detallan a continuación:

- **Tamaño**. La capacidad de un buffer puede resultar clave a la hora de mejorar el rendimiento de un sistema, pero puede también influir negativamente en el consumo de memoria. El parámetro `tamaño` determina el número máximo de objetos `Data` que podrán ser almacenados temporalmente en un buffer, por lo que el tamaño efectivo del buffer dependerá de la naturaleza de los objetos almacenados[4]. Supongamos un caso típico de procesamiento de una base de datos acústica. El primer procesador, que se encargará de cargar uno a uno los ficheros de audio, podría llegar a volcar todo el contenido de cada fichero de audio en un único objeto `IntData`. A continuación podríamos encontrar un procesador que realiza el ventaneo de la señal de audio, pudiendo generar una secuencia compuesta de objetos `DoubleData` a partir de los cuales sucesivas fases de procesamiento podrán extraer algún tipo de información espectral. En este caso concreto, sería recomendable no interconectar los procesadores de carga de ficheros de audio y el de ventaneo con un buffer de mucha capacidad, ya que su tamaño estaría directamente relacionado con el número de ficheros de audio completos que podrán ser almacenados en el buffer. Por el contrario, los siguientes buffers de la cadena de procesamiento podrían perfectamente ser de gran capacidad, ya que el tamaño de los elementos a almacenar es mucho más pequeño.

- **Bloqueo de escritura**. El bloqueo de escritura asegura que en caso de encontrarse lleno el buffer, la operación de inserción por parte del productor quedará bloqueada a la espera de espacio. Si se desactiva el bloqueo de escritura, las operaciones de inserción nunca sufrirán bloqueo alguno, y en caso de tratar de insertar un nuevo elemento en un buffer lleno, el usuario recibirá una notificación de desbordamiento de buffer. En la mayoría de los casos, no tiene sentido alguno desactivar el bloqueo de escritura. Sin embargo, hay casos en los que estamos obligados a hacerlo. Si una cadena de procesamiento parte de una entrada *viva*, por ejemplo un micrófono, el bloqueo de escritura carece de sentido. Cualquier procesador intermedio con una velocidad de procesamiento inferior a la derivada de la adquisición de audio generaría un cuello de botella que conllevaría inexorablemente la pérdida intermitente de la señal adquirida. Es más, esta pérdida de señal no generaría error alguno en la cadena de procesamiento, más allá de que el tamaño de la secuencia final de datos no sería acorde al intervalo de adquisición. En tales casos, resulta más sensato desactivar el bloqueo de escritura y, en caso de desbordamiento de buffer, aumentar la capacidad de los mismos.

Sautrela establece unos valores por defecto para las propiedades de los buffers. En concreto, el tamaño por defecto de un buffer es de 100 y la política de bloqueo de escritura está activada por defecto. Esta configuración es válida para la mayoría de los casos pero, como ya se ha explicado previamente, existen situaciones en las cuales será preciso establecer una configuración específica.

Si bien el lenguaje Java ofrece un conjunto sencillo y claro de herramientas para la creación de aplicaciones concurrentes (multihilo), no es menos cierto que la programación concurrente es una disciplina compleja cuando es preciso establecer mecanismos

[4]Por tamaño efectivo nos referimos a la suma del tamaño real del buffer, que únicamente contiene referencias a los objetos almacenados, y el tamaño propio de dichos objetos.

de sincronización (interconexión) entre los distintos hilos [9]. En las aplicaciones concurrentes, además de coexistir dos o más tareas ejecutándose en paralelo, puede también darse el acceso simultáneo a recursos, bien sean bases de datos, ficheros, *sockets* o variables en memoria. La programación concurrente debe hacer frente a dos problemas: la consistencia de memoria y la exclusión mutua o disputa de hilos (*thread contention*). El primero se refiere al intento simultáneo de actualización de un recurso compartido, mientras que el segundo se refiere a la visibilidad de los cambios realizados sobre un recurso compartido.

3.3.1. Consistencia de memoria

Las arquitecturas hardware actuales, basadas en plataformas multinúcleo y memorias caché de diverso nivel, pueden generar efectos inesperados en lo que a la programación concurrente se refiere. En el caso concreto de las variables, cada hilo de ejecución puede hacer uso de copias locales que serán alojadas en la memoria caché, acelerando el acceso en varios órdenes de magnitud (véase Figura 3.3). Si dicha variable es compartida por varios hilos, pudiera darse el caso de que cada hilo tenga su propia copia local, pudiendo contener valores distintos simultáneamente y, además, por tiempo indeterminado. El error de consistencia de memoria es a menudo denominado *Error de Visibilidad*. En Java, el modificador `volatile` permite definir variables especiales que aseguran la visibilidad de su estado: todo cambio realizado sobre una variable declarada volátil es propagado al resto de las copias. Los mecanismos de exclusión mutua que se describen a continuación también aseguran la visibilidad de una variable. Sin embargo, debido a su elevado coste computacional, no resultan una herramienta adecuada para aquellos casos en los que solo se pretende asegurar la visibilidad de las variables.

3.3.2. Exclusión mutua

La resolución de la disputa por el acceso a escritura de un recurso compartido es, sin lugar a dudas, la operación más costosa en cualquier entorno de programación concurrente. Resulta preciso establecer algún mecanismo de exclusión mutua que coordine los intentos simultáneos de modificación de dichos recursos por parte de los hilos en ejecución. En Java existen dos mecanismos de exclusión mutua, los bloqueos y las operaciones CAS (*Compare And Swap*):

- Los **bloqueos** proporcionan tanto la exclusión mutua como la visibilidad de las variables, pero a costa de un elevado coste computacional. El bloqueo de un recurso implica un arbitraje que típicamente se resuelve mediante un cambio de contexto al kérnel del sistema operativo, el cual suspenderá los hilos que disputan por el acceso a un recurso, a la espera de que el bloqueo sea liberado. Ante un cambio de contexto, el sistema operativo es libre de dar paso a otras tareas ajenas a la propia aplicación, lo que podría provocar la pérdida de los datos e instrucciones almacenados en la memoria caché. Este hecho puede conllevar una reducción sustancial del rendimiento en los microprocesadores modernos [177]. El lenguaje Java ofrece, a través de la *sincronización*, un mecanismo implícito y estructurado de bloqueo y, mediante el paquete `java.util.concurrent.locks`, un mecanismo explícito y no estructurado de bloqueo [9].

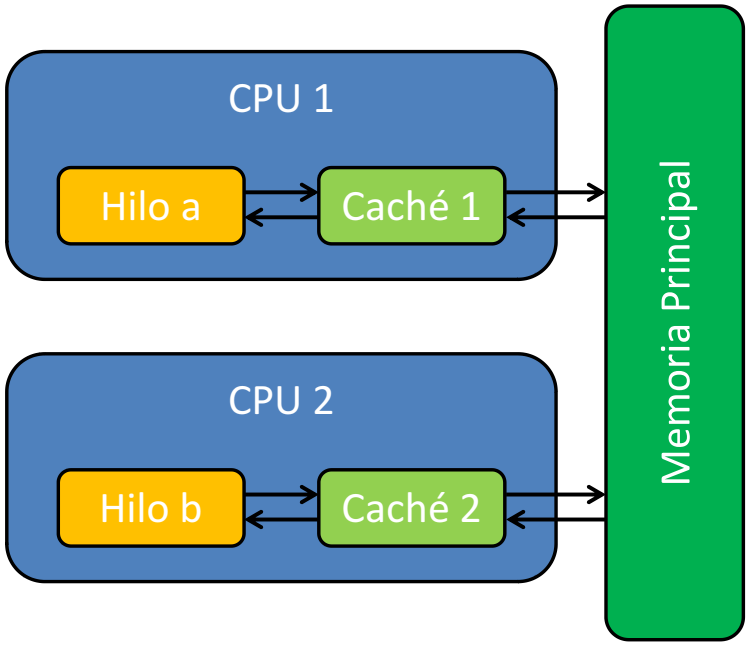


Figura 3.3 Debido a que la caché de cada microprocesador o núcleo puede mantener una copia local de una variable compartida por varios hilos, dichas copias locales no estarán, en principio, sincronizadas. El modificador `volatile` asegura la visibilidad de su estado: todo cambio realizado sobre una variable declarada volátil es propagado al resto de las copias.

- Las **operaciones CAS** están orientadas a la manipulación atómica de variables. Los procesadores modernos cuentan con instrucciones atómicas de chequeo y modificación que proporcionan tanto la exclusión mutua como la visibilidad de los recursos compartidos de manera nativa. Sin embargo, los recursos asociados a dichas instrucciones deben ser de tamaños diminutos, típicamente una word. A diferencia de los bloqueos, las operaciones CAS no conllevan un cambio de contexto, por lo que resultan más adecuadas en aquellas situaciones en las que los recursos a manipular sean informaciones sencillas (contadores, variables booleanas, etc.). Por otro lado, conviene siempre tener en cuenta que una operación CAS no asegura el acceso al recurso, y al finalizar su ejecución informará de si ha resultado exitosa. Si dos hilos compiten por un recurso mediante sendas operaciones CAS, uno conseguirá ejecutar la operación y el otro no, por lo que típicamente las operaciones CAS suelen dar lugar a meta-operaciones basadas en un ciclo que se ejecuta indefinidamente mientras la operación no resulte exitosa. El lenguaje Java ofrece, a través del paquete `java.util.concurrent.atomic`, un conjunto de clases que implementan tanto las operaciones CAS básicas sobre variables sencillas (booleanos, enteros y reales) así como las meta-operaciones asociadas a las mismas.

	Tiempo (ms)	
Método	1 hilo	2 hilos
Referencia	300	—
Variable volátil	4.700	—
Bloqueo	10.000	224.000
CAS	5.700	30.000

Tabla 3.1 Tiempos de ejecución para 500 millones de incrementos de un contador al incluir el coste de diferentes mecanismos de sincronización y el uso de hilos. La ejecución fue realizada sobre un hardware Intel Westmere EP 2.4Ghz. Fuente: [177].

La Tabla 3.1 muestra los tiempos de ejecución de diferentes versiones de una sencilla función que realiza 500 millones de veces el incremento de un contador. Partiendo de los tiempos de ejecución de una versión no concurrente, el mero hecho de asegurar la visibilidad del contador (declarándolo volátil a pesar de no existir concurrencia alguna) aumenta el tiempo de ejecución en un orden de magnitud. El uso del bloqueo ralentiza la ejecución y cuando a esto se le añade la concurrencia de dos hilos, se obtienen los tiempos de ejecución más elevados. Como se ha comentado anteriormente, el caso de un contador que es incrementado en un entorno concurrente encaja perfectamente con las operaciones CAS, y así lo demuestran los tiempos de ejecución. Nótese que de los tiempos de ejecución de la tabla se desprende que frente al caso base (un hilo), el tiempo de ejecución de la versión concurrente más rápida (2 hilos con CAS) resulta dos órdenes de magnitud superior, y el tiempo de ejecución de la más lenta (2 hilos con bloqueo) resulta casi tres órdenes de magnitud superior.

A partir de los valores de la Tabla 3.1 podría surgir la duda de si la concurrencia puede aportar alguna mejora de rendimiento, ya que se podría argumentar que resulta muchísimo más eficiente realizar dos ciclos secuenciales de 500 millones de incrementos (con un tiempo estimado de 600ms) que cualquiera de las dos opciones concurrentes, que requieren en el mejor de los casos 30.000ms. Sin embargo, debemos tener en cuenta dos cuestiones. La primera, que a veces la concurrencia viene impuesta, pudiendo darse el caso de que uno de los dos agentes (el productor o el consumidor) sea un componente externo con el cual debemos intercambiar información y, por tanto, el proceso sea concurrente por naturaleza. La segunda, que el tiempo mostrado en la tabla se refiere únicamente al coste del intercambio de información (el tiempo de acceso y modificación de un recurso compartido). Sin embargo, en un sistema concurrente de procesamiento de la información debemos tener también en cuenta cuánto tiempo se dedica a procesar dicha información. Supongamos que tenemos un sistema concurrente formado por N procesadores y que requiere un tiempo T para realizar la tarea. Supongamos que una fracción del tiempo total, αT $(0 < \alpha < 1)$, ha sido dedicado al intercambio de datos productor-consumidor (cuanto mejor sea la implementación concurrente, menor será el valor del factor α). Por tanto, el tiempo dedicado al procesamientos de datos ha sido $(1 - \alpha)\, T$. Pongámonos ahora en la más optimista de las situaciones para la programación concurrente: ningún procesador ha estado en pausa por falta de datos (no existen cuellos de botella debidos a procesadores lentos) o hilos de ejecución (la

arquitectura soporta la ejecución simultánea de N hilos), es decir, el tiempo dedicado por cada hilo al procesamiento de datos ha sido $(1-\alpha)\,T$. En tal caso, de haber realizado todo el procesamiento de manera secuencial, nos habríamos ahorrado la parte de interconexión de hilos, pero sin embargo el tiempo total para realizar el procesamiento crecería linealmente con el número de procesadores y sería $N\,(1-\alpha)\,T$.

3.3.3. Nociones para el diseño de estructuras eficientes

Como se ha mencionado anteriormente, un buffer sirve de almacenamiento intermedio entre dos procesos, uno productor y otro consumidor. A pesar de no resultar obvio a primera vista y debido a la naturaleza de los productores y consumidores, en la mayoría de los casos los buffers tienden a estar en uno de sus dos estados extremos: vacíos o llenos. Siempre que las velocidades de producción y consumo sean sostenidas y ligeramente distintas (hecho que puede aplicarse a la gran mayoría de los casos de uso), nos encontraremos ante un caso extremo:

- El buffer está vacío. El consumidor está la mayor parte del tiempo esperando a que el productor deposite un dato en el buffer. Si tan pronto el productor deposita un dato en el buffer, ello es notificado al consumidor, este posiblemente procesará el dato antes de que el productor haya depositado el siguiente, volviendo a quedar a la espera.
- El buffer está lleno. El productor está la mayor parte del tiempo esperando a que el consumidor extraiga un dato del buffer (libere espacio del buffer). Si tan pronto el consumidor extrae un dato del buffer, ello es notificado al productor, este depositará un nuevo dato y posiblemente tenga dispuesto otro nuevo antes de que el consumidor haya procesado el que extrajo (y vuelva a extraer otro), quedando el productor a la espera del consumidor nuevamente.

Ambas situaciones extremas resultan simétricas y si no son tenidas en cuenta a la hora de diseñar una estructura de buffer, puede llegarse a la paradójica situación en la que se haya implementado un buffer de longitud N en el que prácticamente siempre tenemos uno o ningún elemento (o de manera simétrica, quede espacio para uno o ningún elemento). No obstante, el no aprovechamiento del espacio de almacenamiento del buffer no es en sí mismo un factor que repercuta en el rendimiento. No debemos olvidar que aquellos productores o consumidores más veloces estarán en constante suspensión y reactivación, afectando seriamente al rendimiento global de la aplicación. Una sencilla manera de minimizar el efecto de los buffers extremos es postergar la notificación de nuevos datos hasta asegurar un mínimo de ocupación (o simétricamente, postergar la notificación de la extracción de datos hasta asegurar un mínimo de espacio disponible). De esta manera, cuando sea notificado el hilo veloz, este contará con un mínimo de espacio en el buffer que le permita realizar más de una operación.

Un segundo factor a tener en cuenta es que, como ha podido comprobarse en la Tabla 3.1, la exclusión mutua repercute muy negativamente en el rendimiento de un sistema. Resulta por ello imprescindible minimizar los puntos de exclusión mutua. En el caso de un buffer, si el acceso al mismo (tanto en inserción como en extracción) es implementado mediante exclusión mutua, el hilo productor y el consumidor estarán en constante disputa por el recurso (el buffer mismo). Por el contrario, es posible

realizar una implementación en la cual el productor pueda consultar en un instante dado el espacio disponible en el buffer, asegurándose así de que, mientras no exceda dicho espacio, es posible realizar operaciones de inserción sin necesidad de mecanismos de exclusión mutua. Equivalentemente, el consumidor puede consultar el número de elementos depositados en el buffer y extraerlos sin necesidad de mecanismos de exclusión mutua. Las implementaciones basadas en zonas de no exclusión pueden llegar a mejorar notablemente el rendimiento de los procesamientos concurrentes.

3.3.4. Implementaciones de referencia

El paquete `java.util.concurrent` cuenta con un conjunto de implementaciones de referencia de estructuras FIFO diseñadas para entornos de programación concurrente:

`ConcurrentLinkedQueue` Se basa en una implementación de lista ligada dinámica (tamaño variable), concurrente y sin bloqueos (hace uso de instrucciones CAS) basada en los algoritmos descritos en [112]. A pesar de tratarse de una implementación interesante en cuanto a su eficiencia, el caso de uso de un buffer de tamaño dinámico no está contemplado en el diseño de Sautrela y, por tanto, queda descartada.

`ArrayBlockingQueue` Se trata de una implementación FIFO estática y concurrente clásica, respaldada por un array de capacidad fija donde se almacenan los elementos. La inserción y extracción de elementos está regida por mecanismos de bloqueo que aseguran la exclusión mutua. Como se ha comentado anteriormente, este tipo de implementaciones acarrean excesivos ciclos de suspensión y reactivación de los hilos de ejecución. Es por ello que queda descartada.

`LinkedBlockingQueue` Se trata de una implementación FIFO estática o dinámica (es posible limitar su capacidad) respaldada por una lista ligada. Al igual que ocurría con la implementación anterior, la exclusión mutua se basa en mecanismos de bloqueo, por lo que no aporta ninguna característica nueva que pueda resultar interesante y, por tanto, queda descartada.

Existen implementaciones de terceros que persiguen la creación de buffers eficientes para los entornos concurrentes. Nos fijaremos en una de ellas:

`Disruptor` LMAX (`http://www.lmax.com`) es una plataforma de negociación en el mercado de divisas orientada a los entornos profesionales, institucionales e interbancarios. Como tal, el tiempo requerido por sus sistemas para cerrar operaciones de compra-venta resulta crucial para su negocio. Las operaciones de compra y venta dependen a su vez de notificaciones de ofertas en tiempo real, las cuales deben ser almacenadas en un buffer a la espera de ser procesadas. Ante este escenario, una implementación eficaz de buffer puede suponer una ingente cantidad de beneficios. El Disruptor es el resultado de los desarrolladores de la compañía en busca de una implementación con baja latencia (retardo en el procesamiento del elemento depositado en el buffer) y alto rendimiento (número de operaciones por segundo) [4, 177]. El diseño se basa principalmente en la obtención por parte de productores y consumidores de zonas de no exclusión en

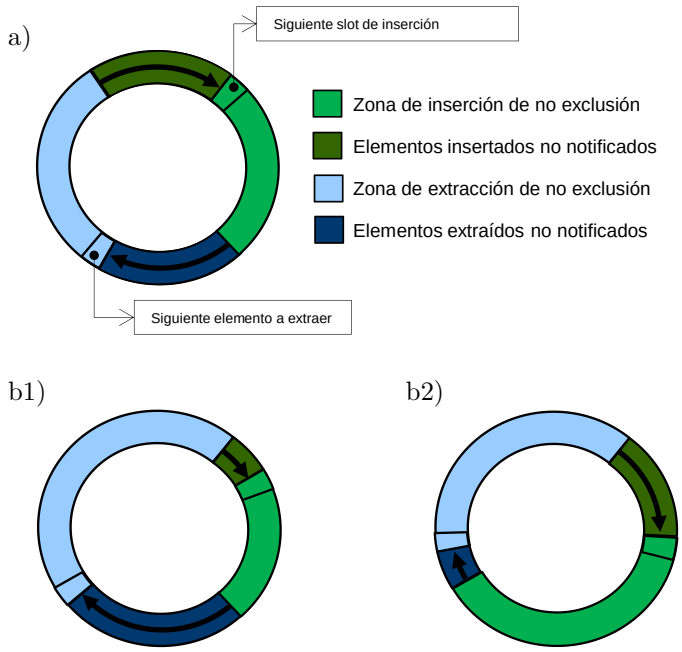


Figura 3.4 Estado y evolución de un buffer en base a la implementación de Sautrela. En un estado cualquiera, a) productor y consumidor cuentan con una zona de trabajo sin exclusión en las cuales las operaciones de inserción y extracción son realizadas sin necesidad de mecanismo de bloqueo alguno. Pasados unos instantes, b1) el productor ha notificado la inserción de un bloque de elementos, aumentando la zona de no exclusión del consumidor. Instantes después, b2) el consumidor notifica la extracción de un bloque de elementos, aumentando la zona de no exclusión del productor.

las cuales poder realizar las acciones que sean precisas sin necesidad de métodos de exclusión mutua. No obstante, el Disruptor ha sido diseñado para entornos de múltiples productores y consumidores, y presenta además una interfaz de uso complicada. A pesar de no haber sido integrado en Sautrela, las ideas principales de su diseño fueron utilizadas en la implementación de los buffers de Sautrela.

3.3.5. Implementación de Sautrela

Los buffers de Sautrela están basados en estructuras circulares estáticas respaldadas por un array. A diferencia de la clase `ArrayBlockingQueue`, la implementación de los buffers de Sautrela se ha basado en el uso de zonas de no exclusión, con el fin último de minimizar las situaciones de exclusión mutua. La Figura 3.4 a) muestra el estado del buffer en un instante cualquiera. Ambos, productor y consumidor, cuentan con una zona de trabajo sin exclusión (de color verde claro para el productor y azul claro para el consumidor). Mientras se limiten a dicha zona, es posible realizar operaciones de inserción y extracción sin necesidad de mecanismo de bloqueo alguno. En vez de notificar cada cambio realizado en el buffer (lo que implicaría al menos asegurar la visibilidad de dicha información), cada hilo espera a haber procesado un bloque mínimo

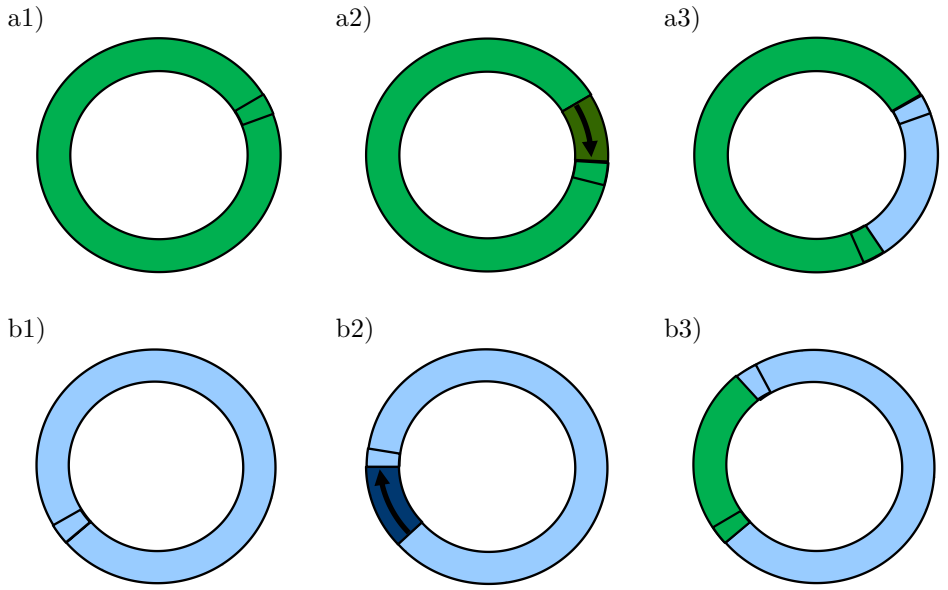


Figura 3.5 Evolución temporal de un buffer vacío (a1, a2 y a3) y otro lleno (b1, b2 y b3). En un buffer vacío, inicialmente todo el buffer es parte de la zona de no exclusión del productor. El hilo consumidor es reactivado cuando existe un número considerable de elementos en el buffer, instante en el que ambos cuentan ya con una zona de no exclusión. En un buffer lleno, inicialmente todo el buffer es parte de la zona de no exclusión del consumidor. El hilo productor es reactivado cuando existe un número considerable de huecos en el buffer, instante en el que ambos cuentan ya con una zona de no exclusión.

de elementos antes de hacerlo. La Figura 3.4 b1) muestra el mismo buffer unos instantes después, cuando el productor ya ha notificado la inserción de un bloque de elementos, aumentando así la zona de no exclusión del consumidor. Algunos instantes después, Figura 3.4 b2), es el consumidor quien notifica la extracción de un bloque de elementos, aumentando la zona de no exclusión del productor.

En un caso ideal en el que ambos hilos procesasen la información a la misma velocidad, el intercambio de información se realizaría sin necesidad de bloqueo alguno, únicamente debiendo actualizar de vez en cuando el valor de un par de variables volátiles (para asegurar así la visibilidad de los cambios realizados). En principio, cuanto mayores sean los tamaños de los bloques definidos, mayor será el rendimiento del sistema. Este diseño añade un coste relacionado con el espacio disponible en el buffer, ya que la notificación por bloques implica el no aprovechamiento de todo su espacio (las zonas de color verde oscuro y azul oscuro no son aprovechadas). Una segunda desventaja consiste en la latencia debida a la notificación por bloques. Sin embargo, y a diferencia del caso de uso del Disruptor, la latencia no supone un problema en el entorno de aplicación de Sautrela.

Si, por el contrario, la velocidad de procesamiento del consumidor fuera más elevada que la del productor, el buffer tendería a estar vacío la mayor parte del tiempo. Una vez más, la notificación por bloques permite definir zonas de no exclusión, no siendo

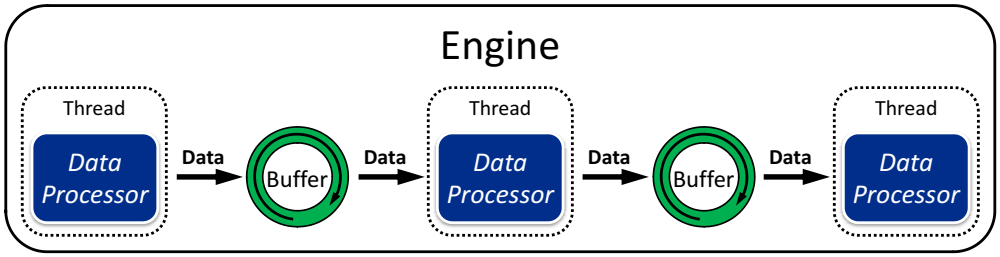


Figura 3.6 Esquema de una *Engine*. Los *Procesadores* realizan la ejecución de sus tareas de manera concurrente, e intercambian los datos a través de los *Buffers*.

necesario reactivar y suspender el hilo consumidor cada vez que el productor ha depositado un elemento en el buffer. Las Figuras 3.5 a1) a2) y a3) muestran la evolución temporal de un buffer vacío. Inicialmente, en a1), todo el buffer es parte de la zona de no exclusión del productor. En a2), el productor realiza la inserción de elementos sin precisar mecanismos de bloqueo. En a3), una vez el productor ha insertado un número suficiente de elementos, este hecho es comunicado al consumidor (su hilo es reactivado), y a partir de ese momento ambos procesos disponen de una zona de no exclusión. Dado que su velocidad es más elevada, el consumidor concluirá de procesar los datos de su zona de exclusión y se quedará a la espera de notificaciones posteriores (el hilo será suspendido), volviendo a la situación original. Las Figuras 3.5 b1) b2) y b3) muestran el caso contrario, el del buffer lleno debido a que la velocidad de procesamiento del productor es más elevada que la del consumidor. Inicialmente, en b1), todo el buffer es parte de la zona de no exclusión del consumidor. En b2), este, realiza la extracción de elementos sin precisar mecanismos de bloqueo. En b3), una vez el consumidor ha extraído un número suficiente de elementos, este hecho es comunicado al productor (su hilo es reactivado), quien a partir de este momento cuenta también con su propia zona de no exclusión. Siendo su velocidad más elevada, el productor concluirá de procesar los datos de su zona de exclusión y se quedará a la espera de notificaciones posteriores (el hilo será suspendido), volviendo a la situación original.

3.4. Engines

Una *Engine* está compuesta por un conjunto de procesadores que son ejecutados en hilos independientes y se mantienen interconectados mediante buffers (véase la Figura 3.6). La ejecución multihilo permite crear aplicaciones de alto rendimiento que aprovechan las prestaciones de las arquitecturas multinúcleo modernas. Si bien es posible pensar en casos de uso más complejos, la ejecución de una engine presenta típicamente las siguientes características:

1. La engine es instanciada y todos los procesadores comienzan su ejecución.

2. Dado que todos los buffers están inicialmente vacíos, todos los procesadores salvo el primero se quedan en suspensión a la espera de datos.

3. El primero de los procesadores de la engine no está conectado a un buffer de entrada. Por el contrario, es el encargado de cargar o adquirir las secuencias de información que serán transmitidas a través de la cadena de procesamiento. La transmisión de información a través de su buffer de salida encadena la reactivación de los procesadores posteriores.

4. El último de los procesadores es conectado a un buffer de salida nulo que descarta toda la información que recibe. Por lo tanto, la función típica de este procesador será la de mostrar o guardar el resultado del procesamiento[5].

5. Una vez finalizada la carga o adquisición, el primer procesador transmite una señal `EOS` y concluye con su tarea.

6. Cuando el siguiente procesador reciba la señal de finalización, concluirá las tareas pendientes, retransmitirá la señal `EOS` y finalizará.

7. Cuando el último procesador finaliza, concluye a su vez la ejecución de la engine.

3.4.1. Instanciación y ejecución de una Engine

Las engines son instanciadas a partir de un documento o descriptor XML que contiene una lista descriptiva de los procesadores y buffers que la conforman. El Listado 3.4 muestra el contenido del fichero `MyEngine.eng` que define una engine ejemplo compuesta por tres procesadores interconectados por dos buffers con capacidad para 1000 elementos y bloqueo de escritura activado. No es obligatorio configurar los buffers, ya que de no hacerlo, toman sus valores por defecto (los nombres son asignados en función de la posición en la engine, el tamaño por defecto es `100` y la política de bloqueo está activada por defecto). Tampoco es preciso dar un valor a cada uno de los parámetros de un procesador, que también pueden tomar sus valores por defecto. Si se desea consultar la documentación de un procesador, puede acudirse al Apéndice C o, como se ha visto en la Sección 3.2.1, ejecutar el comando:

```
$ sautrela -help path.to.mypackage.MyProcessor

PROCESSOR: MyProcessor (path.to.mypackage.MyProcessor)

  This is the documentation of MyProcessor

PARAMETERS:

  myParam1 [int,5] - description of myParam1
  myParam2 [String,"something"] - description of myParam2
  myParam3 [double,1.0] - description of myParam3
  myParam4 [int,10] - description of myParam4
```

[5]De forma análoga al periférico nulo `/dev/null` existente en sistemas operativos tipo Unix, el uso del buffer nulo permite un diseño más sencillo de los procesadores, que realizarán su función independientemente de que se sitúen o no al final de la cadena de procesamiento. Es posible colocar al final de una engine un procesador que no ha sido diseñado para tal fin, situación esta que resulta frecuente durante la fase de diseño de la engine.

Listado 3.4 Fichero `MyEngine.eng`. La engine del ejemplo consta de tres procesadores. Los parámetros que no aparecen toman su valor por defecto.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!DOCTYPE Engine SYSTEM "http://sautrela.org/lib/Engine.dtd">
<Engine name="A symple Engine" description="The description of this Engine">
   <Processor name="path.to.otherpackage.Processor1">
      <param name="proc1Param" value="proc1value" />
   </Processor>
   <Buffer name ="buffer1" size="1000" blocking="true" />
   <Processor name="path.to.mypackage.MyProcessor">
      <param name="myParam1" value="myValue1" />
      <param name="myParam2" value="myValue2" />
      <param name="myParam4" value="myValue4" />
   </Processor>
   <Buffer name ="buffer1" size="1000" blocking="true" />
   <Processor name="path.to.otherpackage.Processor2">
      <param name="proc2param1" value="proc2value1" />
      <param name="proc2param2" value="proc2value2" />
   </Processor>
</Engine>
```

La engine del Listado 3.4 puede ser instanciada mediante la línea de comando:

```
$ sautrela MyEngine.eng
```

Si el descriptor XML no contiene ningún error sintáctico, los nombres de procesadores y parámetros son correctos y los valores proporcionados pueden ser asignados a los parámetros, la engine será instanciada y ejecutada, y el comando finalizará en cuanto concluya la ejecución de la engine. De manera similar a la línea de comando de documentación de procesadores, el descriptor XML de una engine puede utilizar tanto el nombre de clase totalmente cualificado como el correspondiente nombre simple, siempre que no dé lugar a ambigüedad.

La instanciación mediante XML permite la asignación de parámetros sencillos (enteros, reales, booleanos y cadenas de caracteres), tipos de datos enumerados y todo tipo de datos cuyas clases puedan ser instanciadas a partir de una cadena de caracteres (por ejemplo, Ficheros y URLs). El proceso de instanciación de la engine se encarga de comprobar que los valores asignados a los parámetros son compatibles con los declarados en cada procesador.

En el ejemplo del Listado 3.4 los parámetros de los procesadores son establecidos mediante un elemento `param` que contiene dos atributos: `name`, el nombre de parámetro, y `value`, el valor a asignar. También es posible definir el valor a asignar indicándolo dentro del elemento `param`:

```
<param name="myParam">myValue</param>
```

Además, pueden darse situaciones en las cuales el valor asignado a un parámetro interfiera con la propia sintaxis XML. En tales casos es posible definir secciones `CDATA` (Character DATA) que no serán analizadas sintácticamente, y serán tratadas como

Listado 3.5 Fichero `MyEngine2.eng`. Descriptor XML de una engine parametrizada. Dos de los parámetros del segundo procesador podrán ser establecidos al instanciar la engine mediante las opciones de línea de comando `-capacity` y `-color`. Si no se presentan dichas opciones, el primer parámetro tomará el valor `23`, y el segundo su valor por defecto. En este caso se han utilizado nombres simples como identificadores de procesadores.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!DOCTYPE Engine SYSTEM "http://sautrela.org/lib/Engine.dtd">
<Engine name="A symple parameterized Engine" description="The description of this
    Engine">
  <Processor name="Processor1">
    <param name="proc1Param" value="proc1value" />
  </Processor>
  <Buffer size="1000" blocking="true" />
  <Processor name="MyProcessor">
    <param name="myParam1" value="?-capacity [23]" />
    <param name="myParam2" value="?-color" />
    <param name="myParam4" value="myValue4" />
  </Processor>
  <Buffer size="1000" blocking="true" />
  <Processor name="Processor2">
    <param name="proc2param1" value="proc2value1" />
    <param name="proc2param2" value="proc2value2" />
  </Processor>
</Engine>
```

simples cadenas de caracteres. Una sección `CDATA` comienza con la secuencia `<![CDATA[` y finaliza con la secuencia `]]>`:

```
<param name="myParam"> <![CDATA[
this is a <blod> complex </blod> value
]]></param>
```

3.4.2. Parametrización de Engines

Es posible parametrizar una engine de tal forma que los valores de algunos de sus parámetros puedan ser establecidos en el momento de la ejecución. La parametrización de engines es una técnica sencilla y elegante que permite diseñar sistemas configurables en tiempo de ejecución. El Listado 3.5 muestra el descriptor XML de una engine parametrizada. Esta engine permite establecer dos de los parámetros de su segundo procesador mediante opciones de línea de comando:

```
$ sautrela Engine2.eng -capacity 15 -color red
```

Cuando una engine es parametrizada, también es posible redefinir los valores por defecto de los parámetros. En el ejemplo del Listado 3.5, el valor por defecto del parámetro `myParam1` del segundo procesador queda establecido a `23`, de tal manera que si en la línea de comando no se presenta la opción `-capacity`, su valor pasará a ser `23`, independientemente del valor por defecto que tuviera el procesador definido in-

ternamente (originalmente su valor por defecto era `5`). Nótese que los identificadores de opción de una engine parametrizada (en nuestro ejemplo, `-capacity` y `-color`) pueden ser cualesquiera, independientemente del nombre que el procesador haya asignado a dicha propiedad. Dada una engine parametrizada, es posible obtener cierta información sobre sobre la engine y sus parámetros, de manera análoga a la página de manual de un comando de sistema cualquiera:

```
$ sautrela -help Engine2.eng

ENGINE: A symple parameterized Engine

  The description of this Engine

PARAMETERS:

  -capacity [int,23] description of myParam1 [myParam1,MyProcessor]
  -color [String,"something"] description of myParam2 [myParam2,
    MyProcessor]
```

La documentación de una engine está íntimamente ligada a la documentación de los procesadores que la componen. Como puede observarse en el ejemplo, el nombre y descripción de la engine y los nombres de las opciones de línea de comando provienen del descriptor de la engine, mientras que el tipo y la descripción de los parámetros serán aportados por el propio procesador. Los valores por defecto son en principio establecidos por el procesador, pero pueden ser sobrescritos por la engine.

Esta arquitectura de componentes, que permite combinar procesadores y crear engines parametrizadas, junto con la generación automática de páginas de manual, conforma una de las características más destacables de Sautrela. El usuario puede crear diseños de procesos complejos basándose en la reutilización de bloques de procesamiento ya existentes, dando lugar a meta-procesos debidamente documentados y ejecutables a través de comandos parametrizables.

3.4.3. Aplicación *Engine Builder*

Sautrela incorpora una aplicación gráfica, *Engine Builder*, que permite la creación y modificación de engines mediante una sencilla herramienta gráfica. La aplicación, que se muestra en la Figura 3.7, puede ser ejecutada mediante la línea de comando:

```
$ sautrela EngineBuilder
```

La aplicación importa todos los plugins, tanto propios como de terceros, visibles a Sautrela. Cualquier procesador importado puede ser arrastrado y conectado a la línea de procesamiento de la engine, siendo posible modificar cada uno de sus parámetros así como los buffers que los interconectan. Además, la aplicación muestra en todo momento la documentación del procesador seleccionado.

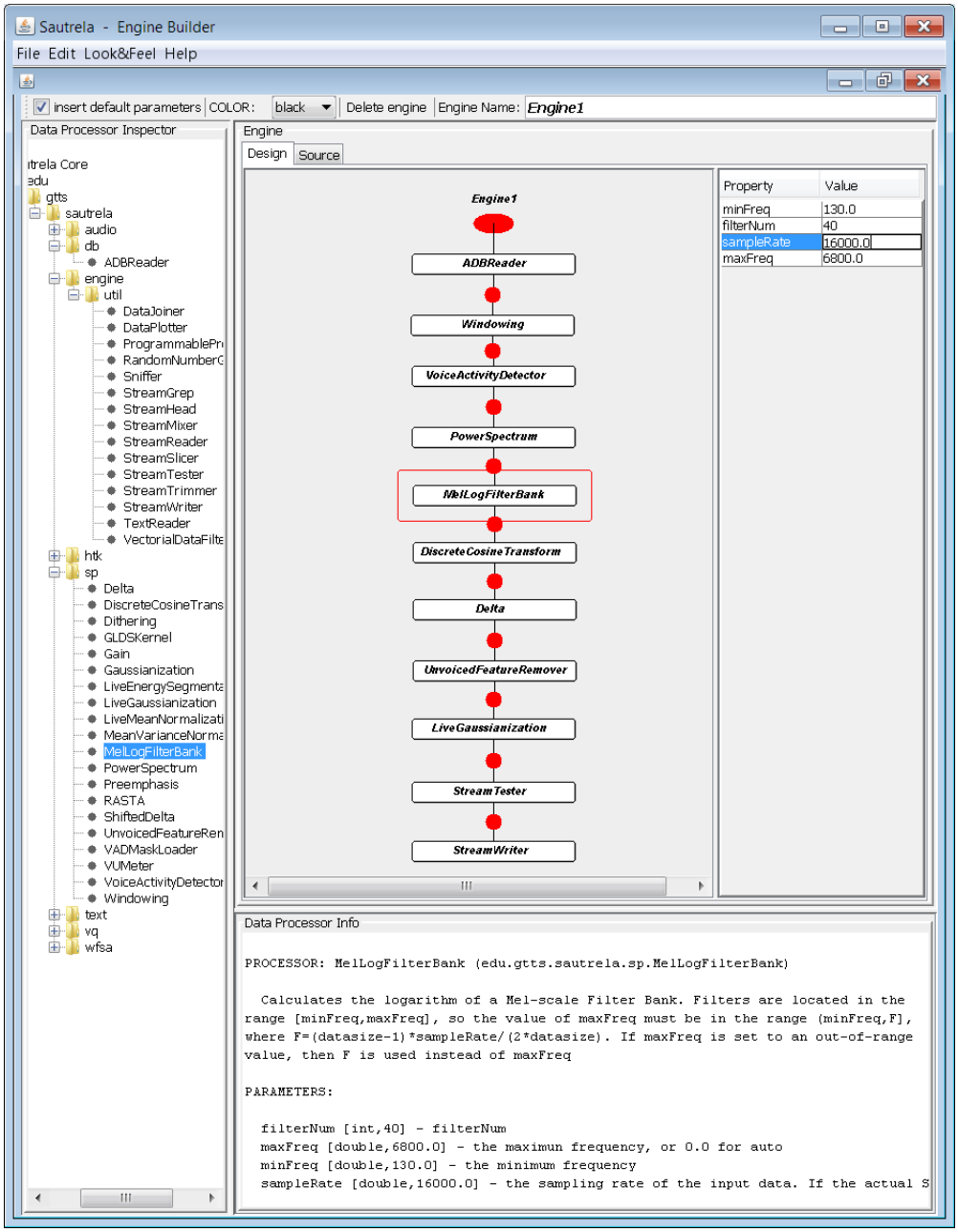


Figura 3.7 La aplicación gráfica *Engine Builder* ofrece una herramienta gráfica para el diseño de engines. Cualquier procesador puede ser arrastrado y conectado a la línea de procesamiento de la engine. Es posible modificar cada uno de los parámetros de los procesadores y los buffers que los interconectan.

3.5. Comandos

Los *Comandos* pueden ser utilizados para crear, inicializar o inspeccionar estructuras de datos relacionadas con los procesadores y las engines (por ejemplo, el chequeo de bases de datos y recursos de audio, la inicialización de modelos, la consulta de procesadores o la creación de engines). Para Sautrela, toda clase ejecutable (aquella que cuente con un método estático `main`) equivale a un comando, y como tal, puede ser ejecutado desde su línea de comando. Si bien Java permite la ejecución directa de dichas clases, el hacerlo a través de Sautrela permite unificar en una sola línea de comando todas las posibilidades de ejecución del entorno de trabajo. De manera equivalente a los procesadores, Sautrela utiliza la introspección para detectar los comandos empaquetados en plugins propios y de terceros, y exige que el fichero MANIFEST.MF contenga una declaración expresa de cada comando:

```
Name: path.to.mypackage.MyCommand.class
Sautrela-Command: True
```

Para ejecutar un comando, solo precisamos el nombre completo de su clase:

```
$ sautrela path.to.mypackage.MyCommand
```

Un vez más, en caso de no dar lugar a ambigüedad, podemos usar el nombre simple:

```
$ sautrela MyCommand
```

De hecho, en anteriores secciones ya se han mostrado ejemplos de comandos: las aplicaciones gráficas *Plugin Navigator* y *Engine Builder* han sido ejecutadas a través de sus correspondientes comandos.

Sautrela establece un sencillo patrón de diseño para integrar la documentación de un comando: la existencia de un método estático `getManPage` que devuelva en una cadena de texto la página de manual del comando. Dicha página de manual podrá obtenerse mediante la línea de comando:

```
$ sautrela -help MyCommand

  MyCommand (path.to.mypackage.MyCommand)

    A short description of this command

  Syntax:  CommandName [-ae] [-i iVal] [-o oVal] arg1 arg2

    -a       This option activates a flag.
    -e       This option activates a flag.
    -i ival  This option takes a value.
    -o oval  This option takes a value.
    arg1     First argument
    arg2     Second argument
```

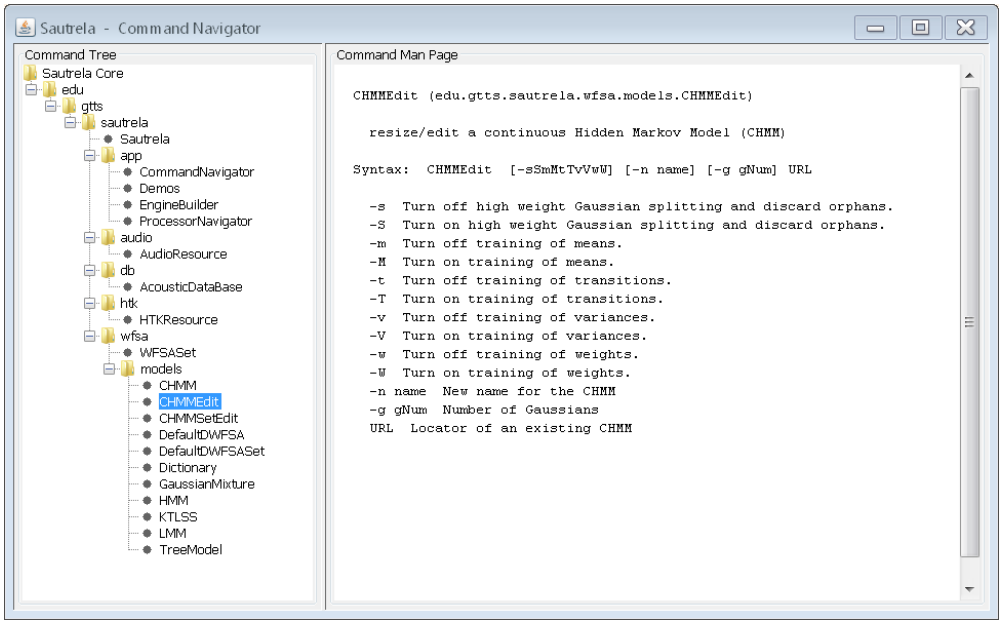


Figura 3.8 La aplicación gráfica *Command Navigator* muestra el conjunto de *Comandos* contenidos en plugins propios y de terceros. Cada comando va acompañado de su correspondiente documentación.

A diferencia de lo que ocurre con los procesadores y las engines, Sautrela no genera la documentación de los comandos, sino que simplemente muestra la documentación que los propios comandos confeccionan (el formato de una página de manual de un comando cualquiera no tiene por qué coincidir con el mostrado aquí).

El Apéndice B contiene una descripción detallada de todos los comandos incluidos en Sautrela. La aplicación gráfica *Command Navigator* (muy similar a la aplicación *Processor Navigator*) permite consultar el conjunto de *Comandos* contenidos en plugins propios y de terceros, así como su correspondiente documentación. La aplicación, que se muestra en la Figura 3.8, puede ser ejecutada mediante la línea de comando:

```
$ sautrela CommandNavigator
```

3.6. Sautrela: línea de comando

Como se ha podido ver en los ejemplos de las secciones anteriores, la línea de comando de Sautrela permite ejecutar *Comandos* y *Engines*, así como obtener páginas de manual de *Comandos*, *Procesadores* y *Engines*. La presente sección describe en profundidad la sintaxis de la línea de comando de Sautrela.

Si ejecutamos el comando `sautrela` únicamente con la opción `-help`, se obtiene la página de manual de la línea de comando:

```
$ sautrela -help

  Usage:  sautrela  [OPTIONS] [<Command|Processor|Engine> [args ...]]

  OPTIONS:

    -help      show the man page of Sautrela/Command/Processor/Engine
    -commonSeed longValue
               change the default common random seed, or "null" to unset it.
    -trace     trace execution
    -version   print product version and exit

  ARGUMENTS:

    Command    the name of a Command
    Processor  the name of a Processor
    Engine     the locator of an Engine
    args       Command/Egine invocation arguments


  Where "sautrela" is an alias of:

    <path_to_java_command> -jar <path_to_Sautrela.jar>
```

3.6.1. Modos de ejecución

Sautrela define dos modos de ejecución: *usuario* y *desarrollador*. El modo por defecto es el de usuario y la opción `-trace` activa el modo desarrollador. La información mostrada ante excepciones (errores) dependerá del modo de ejecución. Así, cuando la ejecución se da en modo usuario, únicamente se muestra una breve información sobre la excepción ocurrida (un fichero no encontrado, un descriptor XML erróneo, un error de formato en un parámetro, etc.). Sin embargo, si el error ocurre en modo desarrollador, es mostrada la traza completa de la excepción ocurrida. Esta sencilla distinción de modos de ejecución ofrece al usuario medio un entorno de ejecución amigable, permitiendo a los desarrolladores acceder a toda la información necesaria para depurar errores.

3.6.2. Control de procesos pseudoaleatorios

Sautrela contiene diversos componentes que hacen uso de fuentes pseudoaleatorias: procesadores para la generación aleatoria de números y símbolos, inicialización aleatoria de modelos, etc. Para que los experimentos puedan ser reproducibles, Sautrela cuenta con un mecanismo de control de la semilla de los procesos pseudoaleatorios. Por defecto, todas las fuentes pseudoaleatorias de Sautrela comparten una semilla común preestablecida. La opción `-commonSeed` permite modificar el valor de dicha semilla común. Al modificar su valor, el resultado de un experimento podrá ser diferente, pero

seguirá siendo reproducible: toda ejecución que mantenga esa misma semilla dará lugar a idénticos resultados. La posibilidad de modificar la semilla de los procesos aleatorios permite cuantificar el efecto que de las fuentes pseudoaleatorias tienen sobre el resultado final.

La asignación de un valor nulo (`-commonSeed null`) desactiva el mecanismo de semilla común y, por tanto, los experimentos dejan de ser reproducibles (cada fuente aleatoria adquiere su propia semilla de manera impredecible).

3.6.3. Documentación

La opción `-help` activa la documentación. En caso de no existir argumento alguno, como ya hemos visto, se muestra el manual del propio comando `sautrela`. En caso contrario, se muestra la página de manual del *Comando*, el *Procesador* o la *Engine* correspondiente. Si se trata de un comando, la página de manual es confeccionada por el propio comando (véase Sección 3.5). Por el contrario, si se trata de un procesador, la página de manual es confeccionada automáticamente (a través de la introspección de la clase que lo representa), ofreciendo información del propio procesador y cada uno de sus parámetros (véase Sección 3.2). Por último, si se trata de una engine, la página de manual es confeccionada automáticamente, pero en este caso la información proviene de diferentes fuentes. El descriptor de la engine determina el nombre y su descripción, así como los parámetros y su correspondiente etiqueta de opción para la línea de comando. Sin embargo, la naturaleza y descripción de dichos parámetros serán determinados por el procesador correspondiente (véase Sección 3.4).

3.6.4. Procesadores, Engines y Comandos

La línea de comando de Sautrela permite interaccionar con tres agentes: *Comandos*, *Procesadores* y *Engines*. En caso de existir un argumento que pudiera referirse a alguno de los tres agentes, dicho argumento es procesado mediante el siguiente procedimiento:

1. Si existe un *Comando* cuyo nombre coincida con el argumento, este es tratado como un *Comando*. De lo contrario, se pasa a la segunda fase.
2. Si la opción `-help` está activa y existe un *Procesador* cuyo nombre coincida con el argumento, este es tratado como un *Procesador* y se muestra su página de manual (los procesadores no son ejecutables y la única opción de línea de comando es acceder a su página de manual). De lo contrario, se pasa a la tercera fase.
3. El argumento es tratado como el localizador (URL) de una *Engine*.

Este orden de procesamiento debe tenerse muy en cuenta en ciertos casos de error en la línea de comando, que podrían dar lugar a equívocos. El caso más típico ocurre cuando el nombre de comando o procesador es erróneo:

```
$ sautrela wrong.command.name

ENGINE instantiation error - FileNotFoundException:

  /home/username/wrong.command.name (No such file or directory)
```

Al no coincidir el argumento con ninguno de los comandos existentes, se descarta que sea un comando. Tampoco se ha activado la opción `-help`, por lo cual ni siquiera se chequeará si se trata de un procesador. Por tanto, el nombre de comando erróneo será tratado como el localizador de una engine. Al tratar de instanciar la engine mediante el localizador `wrong.command.name` (más concretamente `file:wrong.command.name`, tras utilizar el contexto `file:` por defecto), el sistema de archivos generará una excepción, ya que dicho recurso no existe.

Si se activa la opción `-trace`, además de mostrarse la traza del error, se informa de las fases de procesamiento de la línea de comando. Para el caso anterior:

```
$ sautrela -trace wrong.command.name

STEP0 : Command line parsed: trace=true help=false target="wrong.command.
    name" targetArgs=[]

STEP1 : Checking if "wrong.command.name" is a Command --> NO

STEP2 : Checking if "wrong.command.name" is a Processor --> NO

STEP3 : Invoking Engine "wrong.command.name" with arguments "[]"

ENGINE instantiation error - FileNotFoundException:

  java.io.FileNotFoundException: /home/username/wrong.command.name (No
    such file or directory)
    at java.io.FileInputStream.open(Native Method)
    at java.io.FileInputStream.<init>(FileInputStream.java:131)
    at java.io.FileInputStream.<init>(FileInputStream.java:87)
    at sun.net.www.protocol.file.FileURLConnection.connect(
    FileURLConnection.java:90)
    ...
```

3.6.5. Aplicaciones y Demos

El comando `sautrela` puede ser ejecutado también sin argumentos, en cuyo caso se pone en marcha la aplicación gráfica que se muestra en la Figura 3.9. Esta sencilla interfaz ofrece un acceso directo a las *apps* de Sautrela: el conjunto de comandos que cuentan con una interfaz gráfica. Dicho conjunto está formado por:

- ***Processor Navigator***. Aplicación gráfica que muestra el conjunto de *Procesadores* contenidos en plugins propios y de terceros, así como su correspondiente documentación (véase Figura 3.2).
- ***Command Navigator***. Aplicación gráfica que muestra el conjunto de *Comandos* contenidos en plugins propios y de terceros, así como su correspondiente documentación (véase Figura 3.8).
- ***Engine Builder***. Aplicación gráfica para la creación y edición de engines, con acceso a la documentación de cada uno de los procesadores (véase Figura 3.7).

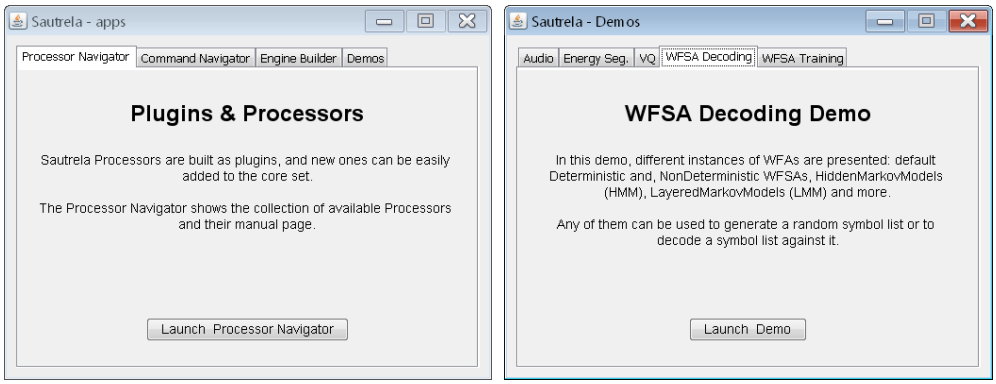


Figura 3.9 La aplicación gráfica *Sautrela* ofrece un acceso directo al conjunto de comandos que cuentan con una interfaz gráfica y que están agrupados en *Aplicaciones* y *Demos*.

- ***Demos***. Aplicación gráfica que muestra un conjunto de demos. Las demos incluidas muestran ejemplos de procesamiento de señales de audio, cuantificación vectorial y entrenamiento y decodificación frente a modelos que implementan la interfaz WFSA (Weighted Finite-State Automata) de Sautrela.

3.7. Características transversales

Durante la creación de procesadores y comandos es común el uso de características o patrones de diseño transversales al conjunto de todos ellos. Casi siempre existen diferentes opciones de implementación, por lo que una buena elección puede dar lugar a componentes que a la postre resulten más versátiles. En la presente sección se analizan algunos de los patrones más interesantes, que podrán encontrarse en muchos de los conjuntos de procesadores que incorpora Sautrela.

3.7.1. Acceso a recursos mediante URLs

A menudo, un procesador o comando debe acceder a algún recurso externo: un archivo de audio, una máscara de voz/no-voz, un fichero parametrizado de datos, un modelo, etc. En tales casos, alguno de los parámetros del procesador o de los argumentos del comando deberá indicar dónde localizar el recurso en cuestión. En Sautrela se ha optado por la solución más versátil: los localizadores uniformes de recursos o URLs [97, 213]. De la misma manera que una ruta de fichero constituye su localizador dentro del sistema de archivos, un URL constituye una referencia a un recurso (no tiene por qué tratarse de un fichero) que puede estar contenido en Internet (o la red). Un caso de uso típico de los URLs se da a diario durante la navegación de páginas web:

`http://es.wikipedia.org/wiki/Localizador_de_recursos_uniforme`

Cuando el esquema de un URL es `http` o `https`, se los suele denominar comúnmente dirección web (localizador de un recurso contenido en la World Wide Web o WWW). El formato general de un URL viene dado por:

`<esquema> :// <autoridad> <ruta> [ ? <pregunta> ] [ #<fragmento> ]`

El esquema de un URL define la manera de acceder al recurso. Existe una infinidad de esquemas diferentes, pero a continuación mostraremos unos pocos que resultan de interés en el entorno de Sautrela:

file Accede a un fichero del sistema de archivos. De hecho, toda ruta tiene su URL equivalente, por lo que el uso de URLs en vez de rutas de fichero no limita el acceso a ficheros locales. Unos ejemplos:

`file://localhost/C:/WINDOWS/Desktop/my.data`

`file:///C:/WINDOWS/Desktop/my.data`

`file:/C:/WINDOWS/Desktop/my.data`

`file://localhost/home/username/data/my.data`

`file:///home/username/data/my.data`

`file:/home/username/data/my.data`

Los primeros tres URLs suponen un sistema operativo Windows y los siguientes un entorno Unix/Linux. Aunque la doble barra que separa el esquema y la autoridad (también denominado host) viene establecido por la especificación de formato de un URL, en la práctica se permite omitirla (como en los ejemplos tercero y sexto).

http o https Accede a un recurso alojado en un servidor web. El recurso puede tratarse de un fichero que reside en el sistema de archivos del servidor o un recurso dinámico que es generado bajo demanda. Un ejemplo:

`http://my.server.com:8080/full/path/showRandomData?maxlength=10`

jar Ofrece acceso a un recurso empaquetado en un fichero JAR (Java ARchive, un formato de archivo basado en ZIP [217] y utilizado en entornos Java) [11]. Este esquema permite localizar un recurso que reside dentro de un fichero JAR:

`jar:file:/C:/WINDOWS/Desktop/my.jar!/path/to/my.data`

Como puede deducirse del ejemplo anterior, es posible a su vez acceder a recursos que residen en un jar remoto:

`jar:http://my.server.com:8080/full/path/my.jar!/path/to/my.data`

Como los ficheros JAR utilizan la codificación ZIP, el esquema `jar:` es compatible con cualquier fichero con formato ZIP:

`jar:file:/C:/WINDOWS/Desktop/my.zip!/path/to/my.data`

Dado que el símbolo `'!'` tiene un significado concreto para los intérpretes de comandos de entornos basados en Unix (GNU/Linux, *BSD, Solaris, etc.), para

URL de contexto	URL	URL resultante
`file:/home/user/`	`dir/my.data`	`file:/home/user/dir/my.data`
`file:/home/user/`	`../dir/my.data`	`file:/home/dir/my.data`
`file:/home/user/`	`/dir/my.data`	`file:/dir/my.data`
`http://sautrela.es/pub/`	`dir/my.data`	`http://sautrela.es/pub/dir/my.data`
`http://sautrela.es/pub/`	`../dir/my.data`	`http://sautrela.es/dir/my.data`
`http://sautrela.es/pub/`	`/dir/my.data`	`http://sautrela.es/dir/my.data`
`jar:file:/tmp/my.jar!/root/`	`dir/my.data`	`jar:file:/tmp/my.jar!/root/dir/my.data`
`jar:file:/tmp/my.jar!/root/`	`../dir/my.data`	`jar:file:/tmp/my.jar!/dir/my.data`
`jar:file:/tmp/my.jar!/root/`	`/dir/my.data`	`jar:file:/tmp/my.jar!/dir/my.data`
`file:/home/user/file.txt`	`dir/my.data`	`file:/home/user/dir/my.data`
`http://sautrela.es/pub/file.txt`	`dir/my.data`	`http://sautrela.es/pub/dir/my.data`
`jar:file:/tmp/my.jar!/root/file.txt`	`dir/my.data`	`jar:file:/tmp/my.jar!/root/dir/my.data`
`file:/home/user/`	`file:/home/my.data`	`file:/home/my.data`
`http://sautrela.es/pub/`	`file:/home/my.data`	`file:/home/my.data`
`jar:file:/tmp/my.jar!/root/`	`file:/home/my.data`	`file:/home/my.data`

Tabla 3.2 Construcción de URLs a partir de URLs de contexto. Cuando el URL comienza con un carácter de barra `"/"`, se descarta la ruta del URL de contexto. Si el URL es absoluto, se descarta por completo el URL de contexto.

poder utilizar el esquema `jar:` en dichos entornos se deben utilizar expresiones literales o bien el carácter de escape `'\'`:

```
'jar:file:/home/username/data/my.jar!/path/to/my.data'
```

```
jar:file:/home/username/data/my.jar\!/path/to/my.data
```

Independientemente del esquema que utilicen, los localizadores pueden ser tanto absolutos como relativos. Un localizador absoluto determina la forma de acceder a un recurso de manera absoluta. Todos los localizadores anteriores, por ejemplo, eran absolutos. Por el contrario, un localizador relativo se basa en algún contexto previo (otro URL) a partir del cual se establece un camino relativo. La Tabla 3.2 muestra diferentes casos de construcción de URLs a partir de URLs de contexto. Si el URL relativo comienza con un carácter de barra `"/"`, entonces se entiende que se trata de una ruta absoluta, descartando la ruta del URL de contexto (únicamente se mantienen el esquema y la autoridad o host del contexto). Si el URL es absoluto, se descarta completamente el contenido del URL de contexto. Tanto los comandos como los procesadores de Sautrela utilizan el contexto por defecto `file:` a la hora de procesar los URLs de sus argumentos o parámetros, permitiendo así utilizar URLs relativas al sistema de archivos por defecto (el usuario no se ve forzado a usar la sintaxis URL a menos que desee expresamente utilizar otro esquema diferente).

El acceso a recursos remotos y empaquetados abre un abanico de posibilidades que, como se verá, resultan muy interesantes a la hora de desarrollar sistemas con Sautrela.

3.7.2. Indicador de selección gráfica de fichero

A menudo, los comandos o procesadores requieren un URL para acceder a algún recurso. En tales casos, resulta interesante ofrecer al usuario la posibilidad de seleccionar el recurso mediante un diálogo gráfico de selección de fichero (*File Open Dialog*). Todos los componentes de Sautrela establecen un atajo o truco mediante el cual se predefine un URL, `file:OPENDIALOG`, que es interpretado como una petición de selección de fichero mediante interfaz gráfica (de hecho, este tiende a ser el valor por defecto). Nótese que este indicador interfiere con el hipotético acceso a un fichero llamado `OPENDIALOG`, en cuyo caso podrían utilizarse los localizadores alternativos `./OPENDIALOG` o `file:./OPENDIALOG`.

3.7.3. XML (Extensible Markup Language)

XML (eXtensible Markup Language) [42, 41, 216] es un lenguaje de marcas derivado del SGML (Standard Generalized Markup Language) [86, 210], desarrollado por el W3C (World Wide Web Consortium) y diseñado para el intercambio de información estructurada de una manera fácil, segura y fiable.

Una de las ventajas de XML es que, al basarse en el repertorio de caracteres Unicode [176, 212], da cobertura a prácticamente todas las lenguas en uso, soportando los conjuntos de caracteres propios de la escritura de cada lengua (actualmente están soportados 123 tipos de escritura diferentes con un inventario de 113.021 caracteres). El basarse en el estándar Unicode no impide que los documentos XML puedan ser codificados de diferentes maneras (codificaciones) a la hora de ser almacenados o transmitidos. El propio estándar Unicode define diferentes esquemas de codificación, tales como `UTF-8`, `UTF-16` y `UTF-32`. También pueden utilizarse codificaciones que solo representen un subconjunto del inventario Unicode (codificaciones *mapeables* a un subconjunto de Unicode), como por ejemplo `ASCII` o `ISO-8859` (con sus variantes, desde `ISO-8859-1` a `ISO-8859-15`), pero en tal caso el documento XML debe forzosamente declarar en su prólogo la codificación utilizada:

```
<?xml version="1.0" encoding="iso-8859-15"?>
```

El uso de XML evita problemas graves de codificación de texto ya que, debido al uso de diferentes entornos de trabajo, es muy normal que distintas codificaciones coexistan en nuestros ficheros de texto. Pasado cierto tiempo desde la creación de un archivo, difícilmente recordaremos cuál era su codificación exacta, lo que a la larga acarreará problemas.

Es posible definir la estructura y sintaxis que un documento XML debe implementar, manteniendo así la consistencia entre los distintos documentos que se refieran a un mismo tipo de información. En el momento de cargar un documento XML es posible realizar una validación que corrobore que el formato es correcto. Además, el prólogo del documento puede contener una declaración expresa de cuál es el formato que implementa, incluyendo una referencia a una definición de tipo de documento o DTD (Document Type Definition):

```
<?xml version="1.0" encoding="iso-8859-15" ?>
<!DOCTYPE Engine SYSTEM "http://sautrela.org/lib/Engine.dtd">
```

Sautrela hace un uso extenso del lenguaje XML en todos aquellos casos en los que se requiera almacenar o transmitir una información estructurada. Tal es el caso de la descripción de las engines, las bases de datos o los modelos.

3.7.4. Expresiones Regulares

Una expresión regular [208], a menudo llamada también *regex*, permite definir un conjunto de cadenas de caracteres mediante un patrón y suele utilizarse para buscar o reconocer cadenas de texto. Las expresiones regulares son representadas mediante una sintaxis propia. En la actualidad coexisten diferentes estándares, tales como *grep*, *Perl*, *Tcl*, *Python*, *PHP* y *awk*. Java da soporte a las expresiones regulares mediante el paquete `java.util.regex`, con una sintaxis muy similar a la de Perl [12] (el estándar Perl sigue evolucionando, pero el conjunto de símbolos definido y su sintaxis se han convertido en un estándar de facto).

Sautrela utiliza expresiones regulares en dos casos de uso principales:

- **Selección o filtrado**: una expresión regular puede servir de filtro de selección de un conjunto de nombres o identidades representadas mediante cadenas de caracteres. Aquellos elementos cuyos nombres coincidan con el patrón indicado, serán seleccionados (o excluidos).
- **Búsqueda de subsecuencias**: una expresión regular puede definir una subsecuencia a localizar en una cadena de caracteres. Por ejemplo, cuando se trocea una cadena de caracteres, es posible definir cuál deseamos que sea el separador.

La presente sección no pretende ser una referencia exhaustiva de la sintaxis de expresiones regulares en Java (en tal caso invitamos al lector a acudir a [12]), pero dado que son parte fundamental de algunos procesadores y comandos, y van a aparecer con asiduidad en la presente memoria, a continuación presentaremos unas breves nociones de dicha sintaxis. Comencemos con unos sencillos ejemplos:

- La expresión regular `hola*` representa a `hol`, `hola`, `holaa`, `holaaa`, etc.
- `co*sa` representa a `csa`, `cosa`, `coosa`, `cooosa`, etc.
- `hola|adiós` representa únicamente a `hola` y `adiós`.
- `(p|m)adre` representa únicamente a `padre` y `madre`.
- `.*` representa a todas la posibles cadenas de caracteres.

De los anteriores ejemplos se deduce que existen ciertos símbolos o caracteres especiales. En efecto, el carácter `'*'` indica una secuencia, el carácter `'|'` indica una alternativa, los caracteres `'('` y `')'` tienen una función delimitadora, y el carácter `'.'` es sinónimo de cualquier carácter. Si bien no entraremos a describirlos, el conjunto de caracteres especiales de las expresiones regulares en Java está compuesto por:

- `'<([{\^-=$!|]})?*+.>'`

Si deseáramos expresar alguno de estos caracteres literalmente, deberíamos utilizar como prefijo el símbolo de escape '\' , tal como sigue:

- `.*` representa a todas la posibles cadenas
- `\.*` representa a `""` (cadena vacía), `"."` , `".."` , `"..."` , `"...."` , etc.
- `\\` representa a `"\"`

Al utilizar la secuencia de escape `'\'`, debemos tener muy en cuenta que Java también la utiliza en la codificación literal de las cadenas de caracteres[6], por lo que si deseamos crear la expresión regular `\\` mediante una cadena de texto literal (el caso de uso en Sautrela), deberemos escribir: `"\\\\"`. Es por ello que en las definiciones anteriores de expresiones regulares se ha evitado el uso de las dobles comillas, que limitaremos a los casos en los que la expresión literal difiera de la cadena que representa.

Los caracteres especiales `'['` y `']'` permiten definir conjuntos de caracteres:

- `[abc]` representa a cualquiera de los caracteres `'a'` , `'b'` o `'c'`.
- `[^abc]` representa a cualquier carácter excepto `'a'` , `'b'` y `'c'`.
- `[a-zA-Z]` representa el conjunto de caracteres contiguos[7] en la codificación, de la `'a'` a la `'z'` y de la `'A'` a la `'Z'`.

Algunos conjuntos ya están predefinidos. Los más significativos son:

- `.` representa cualquier carácter.
- `\d` representa un dígito (`[0-9]`).
- `\D` representa cualquier carácter excepto un dígito (`[^0-9]`).
- `\s` representa cualquier carácter de espaciado o *whitespace* (`[\t\n\x0B\f\r]`).
- `\S` representa cualquier carácter excepto los de espaciado (`[^\t\n\x0B\f\r]`).

El carácter especial `'*'` es un cuantificador, pero no el único (mostramos aquí únicamente la sintaxis denominada *greedy*):

- `X?` significa `X` o nada (la aparición o no de `X`).
- `X*` significa `X` cero o más veces (una secuencia de `X`, aceptando también la secuencia vacía).
- `X+` significa `X` una o más veces (una secuencia no vacía de `X`).
- `X{n}` significa `X` exactamente `n` veces.
- `X{n,}` significa `X` al menos `n` veces.
- `X{n,m}` significa `X` al menos `n` veces pero no más de `m` veces.

[6] Este hecho suele dar lugar a algunos de los errores más típicos y difíciles de localizar en la confección de expresiones regulares literales.

[7] Estos rangos no contienen a los caracteres acentuados propios de la lengua española, por lo que, en caso de pretender darles cobertura, podría utilizarse la expresión regular alternativa `[a-zA-ZáéíóúüñÁÉÍÓÚÜÑ]`.

Ciertos caracteres especiales no "consumen" caracteres (y, por tanto, no representan a carácter alguno), sino que sirven como delimitadores:

- `^` representa el inicio de una línea[8].
- `$` representa el final de una línea.
- `\b` representa el borde de una palabra[9].

Para obtener una información más detallada de la sintaxis de expresiones regulares y los conjuntos definidos en Java, el lector puede acudir a [12]. Si se desea practicar la confección de expresiones regulares, existen herramientas en línea tales como:

- `http://java-regex-tester.appspot.com`
- `http://www.regexplanet.com`

Estas herramientas ofrecen la posibilidad de validar expresiones regulares acordes al estándar de Java.

[8] Como ya se ha visto en ejemplos anteriores, en la definición de conjuntos su significado es diferente, concretamente la negación o el conjunto complementario.

[9] Por ejemplo, la expresión regular literal `"\\bcontrol"` (nótese el doble uso de la secuencia de escape) es localizada en las cadenas `"perder el control"` y `"controlar el cielo"`, pero no así en `"es un descontrol"`.

Capítulo 4

Sautrela: Componentes

Sautrela [122, 123] viene acompañado de un conjunto de plugins que contienen comandos y procesadores que implementan las funcionalidades necesarias para desarrollar un sistema de Reconocimiento Automático del Habla (RAH). A pesar de tratarse de plugins diseñados con una finalidad clara, la naturaleza modular de los componentes permite su reutilización para la creación de otro tipo de sistemas, minimizando así el tiempo y el esfuerzo requeridos. Las secciones que vienen a continuación contienen, cada una de ellas, una breve descripción de un plugin y sus correspondientes componentes: comandos y procesadores. En caso de precisar una descripción más detallada de alguno de los comandos o procesadores de Sautrela, remitimos al lector a los Apéndices B y C, respectivamente.

4.1. Plugin: Utilidades

El paquete `edu.gtts.sautrela.engine.util` contiene un conjunto de procesadores que ofrecen diversas utilidades genéricas que permiten monitorizar, chequear y transformar las secuencias de datos de una engine, así como volcarlas a un fichero o recuperarlas desde un recurso.

4.1.1. Procesadores

Procesador `StreamTester`

Es un analizador que comprueba la integridad de una secuencia de datos y genera un error en caso de producirse alguna de las siguientes condiciones:

- La secuencia de datos no es conforme a:

  ```
  [StreamBegin (IntData|DoubleData|StringData)* StreamEnd]* EOS
  ```

- Un dato transmitido es nulo (`null`).
- Un dato numérico (`IntData` o `DoubleData`) contiene un vector nulo, o alguno de los valores del vector es erróneo ($\pm Inf$ o NaN).

- Un dato de texto (`StringData`) contiene una cadena de caracteres nula.

El comprobador de integridad permite detectar dos tipos principales de errores. Por un lado, los errores de formato, es decir, secuencias incorrectas de datos. Estos errores son debidos a errores de programación en los procesadores, que deben ser corregidos. Por otro lado, los errores de contenido. Estos pueden ser debidos a causas diversas: errores en los procesadores, errores en la configuración de los procesadores o errores en los datos de entrada.

Dado que este sencillo procesador permite detectar a tiempo algunos errores numéricos en las cadenas de procesamiento, es recomendable utilizarlo siempre que se realicen complejos procesamientos de datos. El diseño de procesadores robustos es ciertamente complejo: una señal de audio vacía, o de valor constante, o incluso con una sola componente frecuencial, pueden poner a prueba al mejor de los programadores.

Procesador `Sniffer`

Se trata de un analizador que muestra por la salida estándar las secuencias de datos. Los datos pueden ser mostrados tanto con formato XML[1], como sin formato (solo se muestran los valores, descartándose todos los marcadores `StreamBegin`, `StreamEnd` y `EOS`). Normalmente este procesador es utilizado durante el desarrollo y la depuración de los procesadores y las engines.

Procesador `StreamWriter`

Este analizador vuelca el flujo de datos de una engine a un fichero. El volcado puede realizarse en modo binario (las secuencias de datos son serializadas [209]) o en modo XML (se genera un documento XML que contiene todas las secuencias de datos).

Este procesador presenta principalmente dos casos de uso. Por un lado, ofrece la posibilidad de exportar el resultado de una engine, en cuyo caso optaremos por el modo XML, ya que dicho formato resulta adecuado para el intercambio de información entre distintos sistemas o plataformas[2]. Por otro lado, permite volcar un flujo de datos para ser recuperado posteriormente, en cuyo caso optaremos por el modo binario (si bien es posible utilizar el modo XML para el volcado y recuperación, este modo consume muchos más recursos, tanto computacionales como de almacenamiento). El volcado y recuperación de datos resulta útil en situaciones en las que una engine que contiene una sección inicial constante es ejecutada repetidamente. Por ejemplo, al entrenar modelos acústicos es posible realizar una sola vez la parametrización del conjunto de entrenamiento de la base de datos y volcar la secuencia de datos resultante. Una vez volcada, esa secuencia de datos puede ser usada como entrada de la fase de entrenamiento de los modelos, procedimiento que se repetirá tantas veces como sea preciso.

[1] El texto enviado a la salida estándar no conforma un documento XML, sino que cada dato de las secuencias es convertido a un elemento XML. Para obtener un documento XML, deberá utilizarse el Procesador `StreamWriter`.

[2] Sautrela no incorpora ninguna herramienta de traducción de formatos XML que permita una exportación a ningún otro formato de software de terceros.

Procesador `StreamReader`

Carga un flujo de datos desde un recurso. El flujo de datos puede estar tanto en modo binario como en modo XML. De manera análoga al Procesador `StreamWriter`, este procesador permite importar[3] datos en formato XML o recuperar un flujo de datos binario.

Procesador `StreamHead`

Vuelca a su salida las primeras secuencias de datos o los primeros datos de cada secuencia (o ambas cosas a la vez), descartando el resto de datos o secuencias. Permite descartar todo aquello que no sea el inicio de cada secuencia o las primeras secuencias, por lo que es adecuado para el análisis de las mismas.

Procesador `StreamGrep`

Vuelca a su salida únicamente aquellas secuencias que contengan una propiedad indicada y su valor coincida con el patrón (expresión regular) suministrado, descartando el resto de secuencias. Permite seleccionar el conjunto de secuencias que será procesado posteriormente.

Procesador `StreamSlicer`

Trocea cada secuencia de datos de entrada en secuencias de tamaño fijo, generando tantas nuevas secuencias como sean necesarias. Este procesador puede resultar útil en situaciones en las que no siendo importante la estructura original de las secuencias (no importa si un recurso de audio se transmite como una única secuencia o troceado en diversas secuencias), sí que es importante que las secuencias no sean excesivamente largas.

Procesador `StreamTrimmer`

Descarta muestras al inicio o al final de cada secuencia de datos. Puede darse el caso de que las secuencias de datos contengan bien una cabecera o una cola de datos (o ambas) que no sean relevantes para el procesamiento, o incluso no deban ser procesadas (por ejemplo, segmentos iniciales o finales de silencio). Si el tamaño de los segmentos a descartar es fijo (no varía de una a otra secuencia de datos) y conocido a priori, podemos utilizar este procesador para eliminarlos.

Procesador `StreamMixer`

Carga un recurso que contiene un flujo binario de datos (previamente volcado mediante un Procesador `StreamWriter`) y lo funde con el flujo de datos de entrada. Las secuencias de ambos flujos de datos deben coincidir en longitud y en tipos de datos. La fusión de dos datos numéricos genera un nuevo dato numérico cuyo vector contiene los

[3] Sautrela no incorpora ninguna herramienta de traducción de formatos XML que permita una importación desde ningún otro formato de software de terceros.

valores del dato de entrada y los valores del dato cargado (en ese orden respectivamente), la fusión de dos datos de texto da lugar a la concatenación de ambas cadenas de caracteres, y la fusión de marcadores de inicio y final da como resultado el marcador proveniente de la entrada (los marcadores cargados son descartados). Este procesador permite la confluencia de diversas fuentes de información en una misma cadena de procesamiento. La Sección 4.1.2 muestra un posible caso de uso.

Procesador `VectorialDataFilter`

Filtra los datos vectoriales (`IntData` y `DoubleData`), devolviendo nuevos datos vectoriales que únicamente contienen los campos (posiciones del vector original) indicados. Este procesador permite descartar parte de la información contenida en los datos vectoriales, o incluso duplicarla (el vector resultante contendrá los campos indicados, y si un campo es indexado más de una vez, puede aparecer repetido en el vector resultante). Un caso de uso típico sería descartar parte de la información suministrada por el procesador precedente. Por ejemplo, el Procesador `Delta` que se describe en la Sección 4.5 devuelve un vector compuesto por las componentes estáticas originales, más sus primeras y segundas derivadas. Mediante el Procesador `VectorialDataFilter` es posible quedarse con un subconjunto de la información suministrada (únicamente las componentes estáticas y sus primeras derivadas, o solo las componentes dinámicas, etc.). Nótese que las secuencias de datos originales podrían haber sido previamente volcadas a un fichero de datos, por lo que la combinación del Procesador `StreamWriter` y el Procesador `VectorialDataFilter` hace posible la creación de bifurcaciones en las cadenas de procesamiento, como se muestra en la Sección 4.1.2.

Procesador `DataJoiner`

Agrupa una secuencia de datos generando una secuencia que contiene un único dato. Si los datos son numéricos, genera un único dato numérico con un vector que contiene los valores de todos los vectores. Si los datos son de texto, genera un único dato de texto con la concatenación de todas las cadenas de texto, usando el separador indicado.

Procesador `TextReader`

Genera secuencias de datos textuales a partir de un recurso de texto (una secuencia por línea). Cada línea del recurso es filtrada (eliminando las subsecuencias que coincidan con un patrón indicado), opcionalmente convertida a mayúsculas o minúsculas y posteriormente troceada, para dar lugar a una secuencia de datos textuales.

Procesador `RandomNumberGenerator`

Genera secuencias aleatorias de datos vectoriales, pudiendo configurar el número de secuencias, su longitud, el tipo de dato (`IntData` o `DoubleData`), la dimensión de los mismos y el rango de valores admitidos. Puede resultar útil durante el desarrollo y la depuración de los procesadores y las engines.

Listado 4.1 `CustomEngine.eng` - Descriptor XML de una engine con un procesador programable. Su funcionamiento es muy similar al del procesador `Sniffer`.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!DOCTYPE Engine SYSTEM "http://sautrela.org/lib/Engine.dtd">
<Engine name="RandomNumberGenerator" description="Print to stdout random number
    streams">
  <Processor name="RandomNumberGenerator">
    <param name="dataType" value="?-t"/>
    <param name="maxValue" value="?-M"/>
    <param name="minValue" value="?-m"/>
    <param name="dataDim" value="?-d"/>
    <param name="streamNumber" value="?-s"/>
    <param name="streamLength" value="?-l"/>
  </Processor><Buffer/>
  <Processor name="ProgrammableProcessor">
    <param name="code"><![CDATA[
      Data d = null;
      while ((d = in.read()) != Data.EOS){
        System.out.println(d.toXML());
        out.write(d);
      }
      System.out.println(d.toXML());
      out.write(d);
    ]]></param>
  </Processor>
</Engine>
```

Procesador `DataPlotter`

Obtiene una representación 2D a partir de los datos vectoriales de entrada, seleccionando dos de las dimensiones de los vectores o una sola, en cuyo caso los valores del eje X corresponden al índice del dato y los valores del eje Y corresponden a los valores seleccionados. Puede utilizarse en la fase de desarrollo y depuración de las engines y los procesadores para detectar visualmente anomalías en los resultados.

Procesador `ProgrammableProcessor`

Sin lugar a dudas, se trata del procesador más versátil de Sautrela, ya que permite parametrizar su comportamiento mediante el código fuente Java que le es suministrado. Tiene un único parámetro, `code`, cuyo valor debe corresponder a una implementación del método `process` (el único método que contiene la interfaz `DataProcessor` que debe implementar todo procesador). Al ser instanciado, el código suministrado es compilado, y en caso de no generar ningún error, es ejecutado. El Listado 4.1 muestra el ejemplo de una engine que contiene un procesador programable que muestra por la salida estándar cada uno de los datos que recibe en el buffer de entrada (su funcionamiento es muy similar al del Procesador `Sniffer`). Nótese que el uso de una sección `CDATA` permite escribir el código fuente de Java sin preocuparse de posibles interferencias con la sintaxis XML. El primer procesador de la engine es un sencillo generador de números aleatorios. El siguiente comando muestra la página de manual de la engine:

```
$ sautrela -help CustomEngine.eng

ENGINE: RandomNumberGenerator

  Print to stdout random number streams

PARAMETERS:

  -t [INT|DOUBLE,DOUBLE] type of data to generate [dataType,
    RandomNumberGenerator]
  -M [double,1.0] highest value [maxValue,RandomNumberGenerator]
  -m [double,0.0] lowest value [minValue,RandomNumberGenerator]
  -d [int,1] dimension of the vectors to generate [dataDim,
    RandomNumberGenerator]
  -s [int,5] number of streams to generate [streamNumber,
    RandomNumberGenerator]
  -l [int,10] number of vector per DataStream [streamLength,
    RandomNumberGenerator]
```

Y a continuación se muestra el resultado de una ejecución de la engine:

```
$ sautrela CustomEngine.eng -d 6 -s 3 -l 4 -t INT -M 10 -m 0
<StreamBegin msg="Random Double Data. Dim:6 Length:4"/>
<IntData value="9 5 0 8 3 9"/>
<IntData value="6 2 1 8 2 0"/>
<IntData value="2 7 4 7 0 2"/>
<IntData value="6 0 1 4 6 4"/>
<StreamEnd/>
<StreamBegin msg="Random Double Data. Dim:6 Length:4"/>
<IntData value="1 3 9 4 5 10"/>
<IntData value="3 0 2 5 10 5"/>
<IntData value="8 6 6 3 8 6"/>
<IntData value="5 2 2 6 0 6"/>
<StreamEnd/>
<StreamBegin msg="Random Double Data. Dim:6 Length:4"/>
<IntData value="10 10 7 0 8 1"/>
<IntData value="6 3 4 7 3 5"/>
<IntData value="9 2 3 3 8 8"/>
<IntData value="9 1 9 2 4 3"/>
<StreamEnd/>
<EOS/>
```

El Procesador `ProgrammableProcessor` permite la creación de procesadores muy sencillos sin la necesidad de tener que generar un plugin. No obstante, debe tenerse muy en cuenta que el poblar una engine con infinidad de procesadores programables no triviales entorpece el proceso de diseño de engines y dificulta su depuración, además de ir claramente en contra de la filosofía de diseño de Sautrela. Todo elemento de procesamiento no trivial debería dar lugar a su correspondiente procesador parametrizado y documentado. Los procesadores programables ni son parametrizables, ni cuentan con documentación alguna.

4.1.2. Ejemplo de uso: Ramificación y confluencia de datos

Las engines de Sautrela solo permiten un único procesamiento lineal de las secuencias de datos. No existe posibilidad alguna de crear una bifurcación o ramificación de los datos que pueda ser dirigida a una secuencia paralela de procesamiento para que posteriormente (opcionalmente) confluya de nuevo a una sola línea de procesamiento. Esta limitación podría impedir el uso de ciertas técnicas de modelado que se basen en un *frontend* que fusione diversas parametrizaciones. Sin embargo, mediante la ejecución de múltiples engines, y haciendo uso de algunos de los procesadores presentados en la sección anterior, sí que es posible diseñar procesos de ramificación y confluencia de datos.

La Figura 4.1 muestra un ejemplo conceptual de ramificación y confluencia de datos. Una secuencia de datos puede ser inicialmente volcada a un fichero y posteriormente cargada y filtrada[4] para ser procesada por diferentes engines. El resultado de cada procesamiento puede ser volcado de nuevo a un fichero, para posteriormente fusionar el contenido de los ficheros así generados y proseguir con una única línea de procesamiento. Como puede observarse, la combinación de los Procesadores `StreamWriter`, `StreamReader`, `VectorialDataFilter` y `StreamMixer` permite acometer complejas operaciones de ramificación y confluencia de datos.

Nótese que la primera engine, que vuelca los datos en un fichero temporal, deberá finalizar antes de ejecutar cualquiera de las engines de la segunda fase (que cargan los datos desde ese mismo fichero). Una vez finalizada la ejecución de la primera engine, las tres engines de la segunda fase pueden ser ejecutadas en paralelo. Dado que estas vuelcan sus resultados en sendos ficheros temporales que sirven de entrada de datos a la última engine de la tercera fase, la ejecución de esta última deberá comenzar una vez hayan finalizado las tres anteriores. En un entorno Unix, el siguiente conjunto de comandos permitiría ejecutar el conjunto de engines de la Figura 4.1:

```
$ sautrela Engine.1 -out file.1
$ sautrela Engine.2a -in file.1 -out file.2a &
$ sautrela Engine.2b -in file.1 -out file.2b &
$ sautrela Engine.2c -in file.1 -out file.2c &
$ wait
$ sautrela Engine.3 -in file.2a -mix1 file.2b -mix2 file.2c
$ rm file.1 file.2a file.2b file.2c
```

En los comandos anteriores se presupone que:

- La opción `-in` de las engines permite definir el localizador del recurso de entrada del Procesador `StreamReader` correspondiente.
- La opción `-out` de las engines permite definir el fichero de salida del Procesador `StreamWriter` correspondiente.
- Las opciones `-mix1` y `-mix2` de la última engine hacen referencia a los localiza-

[4] El filtrado de las engines de la segunda fase permite generalizar el tipo de ramificación: cada dato de salida de la primera fase puede ser transmitido parcial o totalmente a cada engine de la segunda fase.

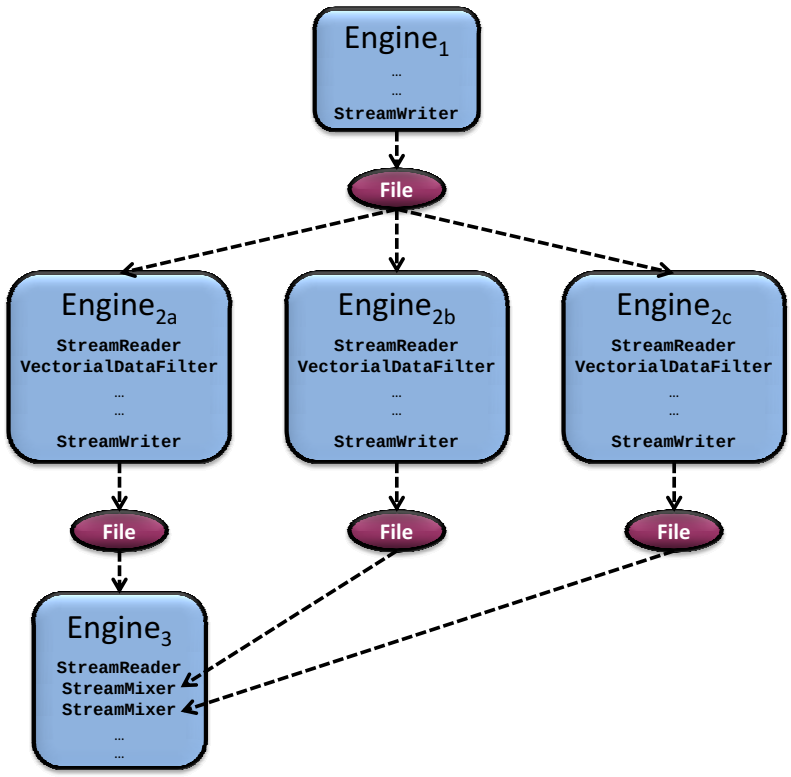


Figura 4.1 Ramificación y confluencia de datos mediante ficheros temporales.

dores de los recursos de entrada de los dos Procesadores `StreamMixer`, respectivamente.

La secuencia de comandos comienza ejecutando la primera de las engines, que vuelca el resultado al fichero `file.1`. El segundo comando inicia la ejecución de la primera de las engines de la segunda fase. El símbolo `'&'` del final del comando permite ejecutar el comando en segundo plano (comienza su ejecución, pero el intérprete de comandos no queda bloqueado a la espera de que concluya). Los siguientes dos comandos inician la ejecución del resto de engines de la segunda fase, también en segundo plano. El comando `wait` espera a que todos los comandos ejecutados en segundo plano (las tres engines de la segunda fase) finalicen, y a continuación se ejecuta la engine de la tercera fase. Finalmente, los ficheros temporales son eliminados.

Es más, es posible ejecutar todas las engines en paralelo. Las denominadas *tuberías nombradas* (*named pipes*) [203] permiten interconectar procesos que usan ficheros intermedios. La tuberías son estructuras FIFO en las cuales un proceso puede realizar operaciones de escritura mientras otro realiza operaciones de lectura (el proceso consumidor no detecta el final de fichero mientras el proceso productor no cierre la tubería). El siguiente conjunto de comandos ejecutaría en paralelo el conjunto completo de engines de la Figura 4.1:

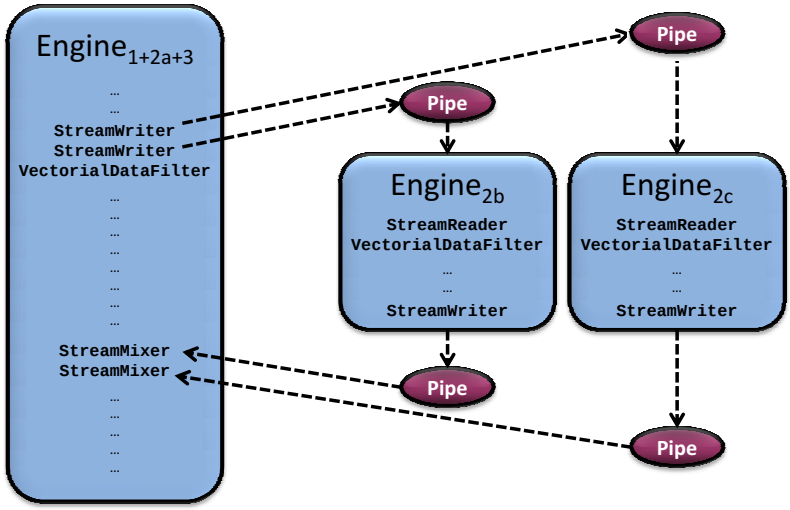


Figura 4.2 Ramificación y confluencia de datos mediante tuberías.

```
$ mkfifo pipe.1 pipe.1.2a pipe.1.2b pipe.1.2c pipe.2a.3 pipe.2b.3 pipe.2c.3
$ tee pipe.1.2a pipe.1.2b > pipe.1.2c < pipe.1 &
$ sautrela Engine.1 -out pipe.1 &
$ sautrela Engine.2a -in pipe.1.2a -out pipe.2a.3 &
$ sautrela Engine.2b -in pipe.1.2b -out pipe.2b.3 &
$ sautrela Engine.2c -in pipe.1.2c -out pipe.2c.3 &
$ sautrela Engine.3 -in pipe.2a.3 -mix1 pipe.2b.3 -mix2 pipe.2c.3 &
$ wait
$ rm pipe.1 pipe.1.2a pipe.1.2b pipe.1.2c pipe.2a.3 pipe.2b.3 pipe.2c.3
```

La secuencia de comandos comienza creando siete tuberías que interconectarán todos los procesos. El segundo comando hace uso del comando `tee`, que copia los datos de entrada en tantos ficheros (tuberías en nuestro caso) como se le indique. De esta manera se logra que la salida de la primera engine (`pipe.1`) vaya a cada una de las tres engines de la segunda fase (`pipe.1.2a`, `pipe.1.2b` y `pipe.1.2c`). El comando es ejecutado en segundo plano para poder seguir con la ejecución de las engines. Los siguientes cinco comandos ejecutan las cinco engines en segundo plano, con las correspondientes tuberías de entrada y salida. El comando `wait` espera a que todos los comandos ejecutados en segundo plano finalicen (en nuestro caso, el comando de redirección `tee` y las cinco engines), y a continuación se eliminan las tuberías creadas al inicio.

El uso de tuberías permite rediseñar el esquema de la Figura 4.1, fusionando las Engines 1, 2a y 3 como se muestra en la Figura 4.2. La secuencia de comandos a ejecutar en este caso será similar a la anterior, algo más simplificada al haber reducido el número de procesos [5]:

[5]Nótese que al concatenar dos Procesadores `StreamWriter` en la primera engine (de ahí las dos

```
$ mkfifo pipe.1.2b pipe.1.2c pipe.2b.3 pipe.2c.3
$ sautrela Engine.1.2a.3 -out1 pipe.1.2b -out2 pipe.1.2c \
    -mix1 pipe.2b.3 -mix2 pipe.2c.3 &
$ sautrela Engine.2b -i pipe.1.2b -o pipe.2b.3 &
$ sautrela Engine.2c -i pipe.1.2c -o pipe.2c.3 &
$ wait
$ rm pipe.1.2b pipe.1.2c pipe.2b.3 pipe.2c.3
```

El ejemplo anterior puede ser simplificado aún más si se hace uso del mecanismo denominado *substitución de modelos* (*process substitution*) [207], que equivale a la creación de *tuberías anónimas* (la entrada o salida estándar de un comando es vista por otro comando como si de un fichero se tratase):

```
$ mkfifo pipe.b pipe.c
$ sautrela Engine.1.2a.3 -out1 pipe.b -out2 pipe.c \
    -mix1 <(sautrela Engine.2b -i pipe.b -o /dev/stdout) \
    -mix2 <(sautrela Engine.2c -i pipe.c -o /dev/stdout)
$ rm pipe.b pipe.c
```

Por último, cabe mencionar que, a pesar de que las tuberías nombradas son creadas en un sistema de archivos convencional, los datos que se escriben en ellas no son volcados al sistema de archivos. Cada tubería cuenta con un buffer en memoria que es utilizado para la escritura/lectura de datos, por lo que, además de no interactuar con el sistema de archivos, las transferencias de datos resultan extremadamente rápidas.

4.2. Plugin: Adquisición y reproducción de Audio

El paquete `edu.gtts.sautrela.audio` contiene un conjunto básico de comandos y procesadores para la lectura, captura y reproducción de audio. Todos los procesadores de este plugin que cargan o adquieren recursos de audio generan secuencias escalares de datos enteros (`IntData`). Además, la información relativa al formato de audio (número de canales, codificación, frecuencia de muestreo y tamaño de muestra, así como propiedades opcionales tales como autor, título, copyright, fecha de publicación, etc.) es guardada en la cabecera (`StreamBegin`) de cada secuencia. La inclusión de propiedades relativas al formato permite que los procesadores sucesivos puedan hacer uso de dicha información. Tal es el caso del procesador para la reproducción de audio, que necesita conocer el formato de audio para realizar su tarea correctamente.

4.2.1. Formatos de audio

Las implementaciones de máquinas virtuales de Java (Java Virtual Machine, JVM) tienden a imponer restricciones con respecto a los formatos de audio soportados. A pesar de que estas restricciones dependen de la JVM utilizada y podrían verse redu-

opciones `-out1` y `-out2`), deja de ser necesario el comando `tee`, ya que las secuencias de datos serán volcadas directamente a cada una de las tuberías.

cidas en sucesivas actualizaciones, Sautrela establece un conjunto básico de formatos soportado por cualquier máquina virtual:

- Formatos de fichero: AIFF, AIFC, AU, SND y WAVE. Los formatos de fichero son contenedores del audio que hacen uso de diversos formatos de compresión y codificación e incluyen cierta meta-información relativa al recurso de audio.
- Formatos de sonido: audio de 8 y 16 bits, en mono y estéreo, con frecuencias de muestreo entre 8 kHz y 48 kHz, y codificaciones lineal, a-law y mu-law.

Dado que la captura o carga de recursos de audio tiene por finalidad el procesamiento de las señales acústicas, los diversos formatos de audio representan un serio problema desde el punto de vista operativo. En principio, sería posible mantener el formato original de las muestras, generando secuencias de bytes o enteros cortos en función de si el audio está codificado en 8 o 16 bits, respectivamente. Por otro lado, si se manejan codificaciones con compresión como a-law o mu-law, cada muestra (de 8 bits) estaría representando realmente una muestra en un rango dinámico superior (16 bits). Incluso en el caso de la codificación lineal pueden darse problemas como la codificación con o sin signo, o el uso de diferentes esquemas de almacenamiento multibyte (sistemas *little-endian* y *big-endian*). Por último, no debemos olvidar que un recurso de audio puede contener varios canales. De mantenerse el formato original del audio, los procesadores sucesivos deberían ser capaces de soportar todo el abanico de posibilidades. Para simplificar esta situación, en aquellos casos en los que se realice la adquisición o carga de un recurso de audio, Sautrela realiza una conversión automática de formato, manteniendo únicamente la frecuencia de muestreo del formato original. Las características del formato de audio resultante son:

- Frecuencia de muestreo: Original
- Tamaño de muestra: 16 bits
- Codificación: lineal PCM con signo
- Canales: 1 (mono)
- Esquema multibyte: *big-endian*

La conversión automática de formato permite a los procesadores abstraerse en cierta medida del formato de audio original y centrarse en la señal acústica que las muestras representan. Una vez realizada la conversión, el audio es volcado a la cadena de procesamiento mediante un flujo escalar[6] de datos enteros (`IntData`). La secuencia de datos estará compuesta por vectores de enteros de tamaños desconocidos y variables en el tiempo. Por ejemplo, un recurso de tres segundos de audio y una frecuencia de muestreo de `16KHz` está compuesto por `48000` muestras. El resultado de cargar dicho recurso podría dar lugar a una secuencia de `48000` vectores de tamaño `1` (un `IntData` por muestra), o `3` vectores de tamaño `16000` (un `IntData` por segundo), o un único

[6]Por flujo escalar de datos (véase Sección 3.1) nos referimos a una secuencia temporal de datos escalares que, no obstante, pueden ir agrupados en vectores (cada vector representa una subsecuencia temporal).

vector de tamaño `48000`, o cualquier combinación de `n` vectores de tamaños diferentes (siempre que la suma de todos los tamaños sea `48000`). Por último, la cabecera `StreamBegin` de cada secuencia de datos contendrá propiedades relativas al formato de audio. En el caso de recursos de audio cargados, serán dos las propiedades: una relativa al formato de fichero original (`AudioFileFormat`) y la otra relativa al formato de sonido convertido (`AudioFormat`). En el caso de la adquisición de audio, la cabecera solo contendrá una propiedad (`AudioFormat`), ya que en este caso ambos formatos coinciden (el procesador de adquisición únicamente permite configurar la frecuencia de muestreo, ya que carecería de sentido permitir configurar el resto de propiedades para a continuación realizar una conversión automática).

4.2.2. Comandos

Comando `AudioResource`

Accede a un conjunto de recursos de audio a través de sus localizadores y, si el formato de audio es compatible, muestra las propiedades de cada uno de los recursos. Este comando está orientado al chequeo de los recursos de audio, ya que por un lado nos informará de si el recurso es accesible (si el esquema del localizador está soportado y existe tal recurso), y por el otro podremos comprobar si el formato de audio está soportado. Nótese que la conversión automática de formato únicamente conserva la frecuencia de muestreo original (un archivo de audio estéreo es compatible con la JVM, pero una vez cargado por un procesador, será convertido a mono).

El siguiente ejemplo muestra la comprobación de tres ficheros de audio. El primero de ellos es un fichero WAV que contiene una señal de 2 segundos en formato estéreo, muestras de 16 bits, codificación lineal y frecuencia de muestreo de 44100Hz. El segundo fichero se ha creado a partir del primero, submuestreándolo a 16KHz y pasando de estéreo a mono. El tercero es un fichero AU con codificación ULAW, obtenido a partir del segundo:

```
$ sautrela AudioResource file1.wav file2.wav file3.au
Resource:  file:file1.wav
  Samples 88200
  SampleSize 16
  SampleRate 44100.0
  Encoding PCM_SIGNED
  Channels 2
  Endianness little-endian
  FileType WAVE

Resource:  file:file2.wav
  Samples 32000
  SampleSize 16
  SampleRate 16000.0
  Encoding PCM_SIGNED
  Channels 1
  Endianness little-endian
  FileType WAVE
```

Listado 4.2 Fichero `PlayAudio.eng`. La engine del ejemplo reproduce el recurso de audio suministrado.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!DOCTYPE Engine SYSTEM "http://sautrela.org/lib/Engine.dtd">
<Engine name="PlayAudio" description="Plays the given Audio Resource">
   <Processor name="AudioResourceReader">
      <param name="audioURL" value="?-i"/>
   </Processor>
   <Buffer/>
   <Processor name="AudioPlayer"/>
</Engine>
```

```
Resource:  file:file3.au
  Samples 32000
  SampleSize 8
  SampleRate 16000.0
  Encoding ULAW
  Channels 1
  Endianness big-endian
  FileType AU
```

4.2.3. Procesadores

Procesador `AudioResourceReader`

Carga un único recurso de audio a través de su localizador, realizando la conversión automática de formato.

Procesador `AudioRecorder`

Realiza una captura de audio a través de la entrada de audio por defecto del sistema, pudiendo configurar la frecuencia de muestreo. Cuenta además con una sencilla interfaz gráfica que permite iniciar y pausar la captura. El procesamiento no concluye hasta que la ventana de la interfaz gráfica sea cerrada.

Procesador `AudioPlayer`

Reproduce el audio que recibe en el buffer de entrada a través de la salida de audio por defecto del sistema.

4.2.4. Ejemplo de uso: reproducción de un recurso de audio

El Listado 4.2 muestra una sencilla engine que reproduce un recurso de audio haciendo uso de dos de los procesadores descritos. El primero carga el recurso de audio suministrado y el segundo lo reproduce. A continuación se muestra la página de manual de la engine:

```
$ sautrela -help PlayAudio.eng

ENGINE: PlayAudio

  Plays the given Audio Resource

PARAMETERS:

  -i [URL,file:OPENDIALOG] the locator of the audio resource or "file:
    OPENDIALOG" for a File Open Dialog [audioURL,AudioResourceReader]
```

4.3. Plugin: Compatibilidad con HTK

El paquete `edu.gtts.sautrela.htk` ofrece la posibilidad de cargar recursos de datos en formato HTK [227], permitiendo el uso de un *frontend* que genere parámetros en dicho formato. El paquete será útil en situaciones en las que sea necesario importar ciertos parámetros que no puedan obtenerse mediante el conjunto de procesadores de Sautrela, o también si se desea hacer una comparación de distintas técnicas de modelado sobre un conjunto común de parámetros de referencia. El plugin no es compatible con recursos que hagan uso de las características de *Compresión* y *Chequeo de CRC* de HTK.

4.3.1. Comandos

Comando `HTKResource`

Accede a un conjunto de recursos de datos en formato HTK a través de sus localizadores y, si el formato es compatible, muestra las propiedades de cada uno de los recursos: el tipo de parámetro, el número total de muestras, el periodo de muestreo y la dimensión de las muestras. El siguiente ejemplo muestra la información de un fichero de datos en formato HTK:

```
$ sautrela HTKResource file.mfcc
  ParameterKind MFCC
  SampleDim 13
  SamplePeriod 100000
  Samples 25228
```

4.3.2. Procesadores

Procesador `HTKResourceReader`

Permite cargar un recurso de datos en formato HTK a través de un URL. Los datos son volcados a la cadena de procesamiento mediante una secuencia vectorial de datos reales (`DoubleData`) y en la cabecera de cada secuencia se almacenan las propiedades del recurso.

4.4. Plugin: Base de datos acústica

El paquete `edu.gtts.sautrela.db` define una estructura de base de datos acústica que permite agrupar toda la meta-información (localizadores de los recursos acústicos, transcripción ortográfica, transcripción fonética, identidad de locutor, etc.) en un único descriptor XML. Ofrece además un comando para inspeccionar el contenido de una base de datos acústica y un procesador capaz de cargar los recursos incluidos en ella. El Listado 4.3 muestra el ejemplo de un descriptor XML de una base de datos acústica compuesta por una secuencia de elementos `Stream`. Cada elemento `Stream` lleva asociado un identificador único, y puede contener un localizador bien de un recurso de audio o bien de un recurso de datos con formato HTK. Cada elemento `Stream` podrá contener una serie de propiedades, consistentes en pares nombre-valor, que conforman la meta-información asociada a cada recurso. En la base de datos del ejemplo, las propiedades son utilizadas para almacenar una transcripción ortográfica, una transcripción fonética y la identidad del locutor. El formato de base de datos acústica no impone restricción alguna a las propiedades que un usuario pudiera añadir.

Los localizadores de los recursos pueden ser tanto absolutos como relativos, ya que el localizador del fichero descriptor XML se usará como contexto a la hora de resolver los URLs de cada recurso. Este simple mecanismo permite generar una base de datos reubicable, siempre que el descriptor XML sea reubicado solidariamente al conjunto de recursos. Dado que el localizador del descriptor XML sirve de contexto, si todos los URLs de la base de datos son relativos, será el localizador del descriptor XML el que informe de la nueva ubicación de la base de datos. Gracias a los diferentes esquemas que pueden ser utilizados en los URLs, la reubicación permite:

- Mover la base de datos dentro del sistema local de archivos. Se trata de la reubicación más sencilla. Suponiendo que el localizador del descriptor del Listado 4.3 fuera:

  `file:/home/user/MyDatabase/MyAcousticDataBase.xml`

  La base de datos estaría contenida en el directorio `/home/user/MyDatabase`. Si moviéramos el subdirectorio `MyDatabase` a cualquier otra ubicación, no se requeriría modificación alguna en el descriptor. El nuevo localizador del descriptor podría ser:

  `file:/new/path/to/MyDatabase/MyAcousticDataBase.xml`

- Alojar la base de datos en un servidor web. Haciendo uso del esquema `http` o `https`, sería posible depositar todo el contenido en un servidor web, de tal manera que el nuevo localizador del descriptor podría ser:

  `http://my.server.com/pub/MyDatabase/MyAcousticDataBase.xml`

- Empaquetar la base de datos. El esquema `jar:` permite empaquetar toda la base de datos en un solo fichero JAR o ZIP, facilitando su manipulación y distribución, y reduciendo el requerimiento de espacio. Es más, el uso de archivos empaquetados suele mejorar los tiempos de acceso a recursos, ya que se reducen las llamadas al sistema de archivos. El nuevo localizador del descriptor podría ser:

Listado 4.3 `MyAcousticDataBase.xml` - Descriptor XML de una Base de Datos acústica. Cada elemento `Stream` puede estar ligado a través de un URL a un recurso, el cual puede contener audio o datos en formato HTK. Las propiedades consisten en tuplas nombre-valor sin restricción alguna (son determinadas por el usuario).

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!DOCTYPE AcousticDataBase SYSTEM "http://sautrela.org/lib/AcousticDataBase.dtd">
<AcousticDataBase name="my Acoustic Data Base">
   <Resource name="wav_ac_train_001_spk_001">
      <Audio url="WAV/sentence001.wav"/>
      <Property name="Ortho" value="hasta la vista"/>
      <Property name="Phon" value="a s t a l a b i s t a"/>
      <Property name="Speaker" value="001"/>
   </Resource>
   <Resource name="wav_ac_eval_001_spk_013">
      <Audio url="WAV/sentence002.wav"/>
      <Property name="Ortho" value="mañana toca excursión"/>
      <Property name="Phon" value="m a h a n a t o k a e s k u r s i o n"/>
      <Property name="Speaker" value="013"/>
   </Resource>
   <Resource name="wav_eval_001_spk_042">
      <Audio url="WAV/sentence381.wav"/>
      <Property name="Ortho" value="pedí otra habitación"/>
      <Property name="Phon" value="p e d i o t r a b i t a z i o n"/>
      <Property name="Speaker" value="042"/>
   </Resource>
   <Resource name="htkmfcc_ac_train_001_spk_001">
      <Htk url="HTK/sentence001.mfc"/>
      <Property name="Ortho" value="hasta la vista"/>
      <Property name="Phon" value="a s t a l a b i s t a"/>
      <Property name="Speaker" value="001"/>
   </Resource>
   <Resource name="htkmfcc_ac_eval_001_spk_013">
      <Htk url="HTK/sentence002.mfc"/>
      <Property name="Ortho" value="mañana toca excursión"/>
      <Property name="Phon" value="m a h a n a t o k a e s k u r s i o n"/>
      <Property name="Speaker" value="013"/>
   </Resource>
   <Resource name="htkmfcc_eval_001_spk_042">
      <Htk url="HTK/sentence381.mfc"/>
      <Property name="Ortho" value="pedí otra habitación"/>
      <Property name="Phon" value="p e d i o t r a b i t a z i o n"/>
      <Property name="Speaker" value="042"/>
   </Resource>
</AcousticDataBase>
```

```
jar:file://home/user/MyDatabase.jar!/MyAcousticDataBase.xml
```

Nótese que el esquema `jar` es también compatible con recursos web, por lo que la base de datos empaquetada podría residir también en un servidor web:

```
jar:http://my.server.com/pub/MyDatabase.zip!/MyAcousticDataBase.
xml
```

Como se verá en los siguientes apartados, es posible acceder a los recursos de una base de datos a través de una expresión regular, lo que permitirá seleccionar el subconjunto de recursos cuyo identificador sea representado por el patrón suministrado. Es recomendable, por ello, que el identificador de recurso se forme de acuerdo a algún esquema que contenga toda aquella información que pudiera ser susceptible de servir como filtro. Supongamos que los identificadores del descriptor del Listado 4.3 hubieran sido creados según el siguiente esquema:

```
[wav|htkmfcc]_[ac_train|ac_eval|eval]_[sentenceID]_spk_[speakerID]
```

En este esquema, "`wav|htkmfcc`" haría referencia a si se trata a un recurso de audio o parámetros en formato HTK, "`ac_train|ac_eval|eval`" se referiría a si el recurso forma parte del subconjunto de entrenamiento acústico, el de evaluación acústica o el de evaluación global, "`sentenceID`" sería un identificador numérico de frase y "`speakerID`" sería un identificador numérico de locutor. En tal caso, las siguientes expresiones regulares permitirían definir los subconjuntos que se muestran a continuación:

- `"wav_.*"` : Todos los recursos de audio (no HTK).
- `"wav_ac_train.*"` : Los recursos de audio para el entrenamiento de modelos acústicos.
- `"wav_ac_eval.*"` : Los recursos de audio de la evaluación de modelos acústicos.
- `"wav_eval.*"` : Los recursos de audio de la evaluación global.
- `"wav_.*spk_042"` : Los recursos de audio del locutor 042.
- `"wav_.*_0.._spk_.*"` : Los recursos de audio de frases con id inferior a 100.

Una segunda forma de acceder a los recursos de una base de datos será mediante el uso de un fichero índice que contenga los identificadores de los recursos a extraer. Este segundo método ofrece una mayor flexibilidad a la hora de crear el descriptor de una base de datos, ya que los identificadores de recursos no tendrían por qué contener información específica alguna (sería perfectamente posible utilizar simples índices numéricos como identificadores de recursos). Sin embargo, esta supuesta flexibilidad es conseguida a costa de gestionar externamente (fuera del descriptor XML) cierta meta-información que es posteriormente utilizada para confeccionar el fichero índice.

Debe tenerse en cuenta también que el orden de extracción de los recursos es diferente para cada método de selección. La selección mediante expresión regular respeta el orden del descriptor de la base de datos (el descriptor es recorrido secuencialmente, extrayendo aquellos recursos representados por el patrón suministrado). Por el contrario, la selección mediante fichero índice respeta el orden del fichero índice, que será

recorrido secuencialmente, extrayendo los correspondientes recursos de la base de datos. En este sentido, si el fichero índice contiene un identificador de recurso inexistente, se generará un error.

4.4.1. Comandos

Comando `AcousticDataBase`

Permite inspeccionar y comprobar el contenido de una base de datos acústica, enviando a la salida estándar la información relativa al subconjunto de recursos definido mediante un fichero índice (muestra únicamente los recursos cuyo identificador esté contenido en el índice) o mediante una expresión regular (muestra únicamente los recursos cuyo identificador quede representado por la expresión regular). El comando permite seleccionar la información que deseamos extraer (el identificador, el localizador y cada una de las propiedades), así como comprobar que todos los recursos estén accesibles (comprueba uno a uno el acceso a todos los recursos a partir del localizador indicado). A continuación se muestra su página de manual:

```
$ sautrela -help AcousticDataBase

  AcousticDataBase (edu.gtts.sautrela.db.AcousticDataBase)

    show content of an AcousticDataBase

  Syntax:  AcousticDataBase  [-cnu] [-e enc] [-p list] <-i URL | -r RegExp
    > URL

    -c  Check (try opening) Audio/HTK Resources
    -n  Don't show Resource name
    -u  Don't show Resource URLs
    -e enc  Output encoding (default: windows-1252)
    -p list  A comma separated list of property names (default:ALL)
    -i URL  Locator of a Resource index (one ID per line)
    -r RegExp  Regular expression for Resource selection
    URL  Locator of an AcousticDataBase
```

4.4.2. Procesadores

Procesador `ADBReader`

Carga desde una base de datos acústica el subconjunto de recursos definido mediante un fichero índice o una expresión regular. Cada recurso genera una secuencia de datos de naturaleza escalar o vectorial, en función de que se trate de un recurso de audio o un recurso de datos en formato HTK, respectivamente. Tanto las propiedades definidas en el descriptor XML como las relativas al formato del recurso son almacenadas en la cabecera de cada secuencia, para que los procesadores sucesivos puedan acceder a dicha información.

4.4.3. Ejemplo de uso: extracción de información

El comando `AcousticDataBase` puede resultar interesante no solo para inspeccionar una base de datos acústica, sino también para extraer información de ella desde la línea de comando. Supongamos que deseamos obtener el vocabulario de la base de datos del Listado 4.3, que no es otro que el conjunto de palabras contenidas en las propiedades etiquetadas como `Ortho`. La siguiente línea de comando extrae las transcripciones ortográficas de todos los recursos de la base de datos:

```
$ sautrela AcousticDataBase -nu -e UTF-8 -p Ortho -r ".*"
    MyAcousticDataBase.xml
 Ortho: hasta la vista
 Ortho: mañana toca excursión
 Ortho: pedí otra habitación
 Ortho: hasta la vista
 Ortho: mañana toca excursión
 Ortho: pedí otra habitación
```

La salida del comando puede ser fácilmente postprocesada para obtener el vocabulario. Una característica interesante de las bases de datos acústicas es que al basarse en XML, se elimina la ambigüedad relativa a la codificación de texto, lo que permite la recodificación libre de errores. En el ejemplo anterior, la salida se obtiene en `UTF-8` independientemente de la codificación en la que esté basado el XML. Equivalentemente, podríamos obtener el inventario fonético a partir del conjunto de transcripciones fonéticas[7]:

```
$ sautrela AcousticDataBase -nu -p Phon -r ".*" MyAcousticDataBase.xml
 Phon: a s t a l a b i s t a
 Phon: m a h a n a t o k a e s k u r s i o n
 Phon: p e d i o t r a b i t a z i o n
 Phon: a s t a l a b i s t a
 Phon: m a h a n a t o k a e s k u r s i o n
 Phon: p e d i o t r a b i t a z i o n
```

4.5. Plugin: Procesamiento de señal

El paquete `edu.gtts.sautrela.sp` agrupa el conjunto de procesadores para el procesamiento de señal que suelen ser utilizados en la fase de parametrización de la señal de voz. Todos estos componentes son configurables, y sus valores por defecto han sido escogidos tomando como referencia una parametrización típica para un sistema de reconocimiento del habla con recursos de audio de 16 kHz y grabados en condiciones de laboratorio.

[7] Nótese que el formato de base de datos acústica no establece que deban existir transcripciones ortográficas o fonéticas, ni que, en caso de existir, deban estar representadas mediante propiedades llamadas `Ortho` o `Phon`. Esta es una decisión de diseño del usuario de Sautrela y, por tanto, las líneas de comando anteriores son únicamente válidas para la base de datos del Listado 4.3.

4.5.1. Procesadores

Procesador `Gain`

Se trata de un sencillo procesador que aplica una ganancia lineal a la señal de entrada.

Procesador `Dithering`

Añade una pequeño valor aleatorio a las muestras de la entrada. El *dithering* es una técnica que evita problemas numéricos en casos en los que las señales de entrada son constantes o tienen componentes frecuenciales constantes. Al añadir un pequeño ruido a las muestras, se evitan valores constantes que a la postre suelen dar lugar a errores de estimación (por ejemplo, varianzas nulas).

Procesador `Preemphasis`

Se trata de un filtro FIR pasa-alto que trata de compensar la atenuación sufrida por las altas frecuencias durante la adquisición del audio. Suele ser utilizado en la primera fase de la parametrización, previa al ventaneo.

Procesador `LiveEnergySegmentator`

Extrae segmentos de señal que contengan una energía superior a un umbral relativo al máximo, descartando los segmentos de baja energía. Dado que el máximo se refiere al máximo valor de energía posible, el segmentador presupone que el rango dinámico de la señal ha sido debidamente ajustado.

Procesador `Windowing`

Realiza el ventaneo de los datos de entrada, pudiendo aplicar diversas funciones de ventana, tales como Rectangular, Hamming, Hanning, Blackman y Blackman-Harris. Se trata del único procesador de Sautrela que convierte una secuencia escalar (vectores de longitud variable que representan secuencias temporales) en una secuencia vectorial (vectores de longitud constante que representan un conjunto de características derivadas de una ventana temporal).

Procesador `VoiceActivityDetector`

Se trata de un detector de actividad de voz basado en energía que procesa señales ya ventaneadas. Dado el máximo de energía de toda una secuencia de ventanas de audio, genera una máscara booleana de actividad de voz relativa a cada una de las ventanas. Aquellas ventanas cuya energía supere un umbral relativo al máximo son clasificadas como voz. Una vez procesada por completo la secuencia de entrada, vuelca esa misma secuencia a la salida, pero habiéndole añadido la máscara de voz a la cabecera.

Así pues, este procesador no altera la secuencia de datos de entrada, únicamente añade la máscara de voz a la cabecera. El Procesador `UnvoicedFeatureRemover` permite eliminar posteriormente aquellos vectores que no contengan voz (según la

máscara añadida). La utilización de una máscara que posteriormente pueda aplicarse para filtrar la salida permite estimar dicha máscara en una fase inicial, sin necesidad de eliminar en ese momento las ventanas que no contienen voz. Nótese que ciertas fases de procesamiento podrían requerir que los vectores suministrados correspondan a instantes consecutivos (derivadas, RASTA, etc.), por lo que es preciso mantener íntegra la secuencia temporal hasta que se hayan superado dichas fases de procesamiento.

Procesador `VADMaskLoader`

Carga las máscaras de actividad de voz desde un fichero de datos (recurso de secuencias de datos en formato Sautrela, véase el Procesador `StreamWriter` del plugin de utilidades, Sección 4.1). Gracias a este procesador, es posible utilizar máscaras de voz/no-voz generadas de manera distinta a como son obtenidas por el Procesador `VoiceActivityDetector`, o incluso importar máscaras de voz/no-voz generadas por software de terceros.

Procesador `UnvoicedFeatureRemover`

Toma de la cabecera la máscara de actividad de voz/no-voz y filtra los vectores de entrada, dejando pasar únicamente aquellos que correspondan a voz.

Procesador `PowerSpectrum`

Calcula el espectro de potencia para cada vector real de entrada. El tamaño de la FFT (Fast Fourier Transform) a aplicar es configurable y debe ser una potencia de 2. Si dicho tamaño es superior al tamaño de los vectores de entrada, la ventana de análisis es rellenada con muestras de valor 0. El tamaño del vector de salida corresponde a la mitad del tamaño de la FFT aplicada.

Procesador `MelLogFilterBank`

Obtiene el logaritmo de la salida de un banco de filtros triangulares basado en la escala Mel [189]. Este banco de filtros, aplicado sobre los vectores que representan el espectro de potencia, tiene una doble finalidad. Por un lado los filtros triangulares suavizan el espectro de potencia y reducen la dimensionalidad de los vectores de salida (la dimensión de los vectores de salida corresponde al tamaño del banco de filtros). Por otro lado, la disposición de los filtros viene dada por la escala Mel, una escala de percepción de la altura (*pitch*) de los sonidos que trata de imitar el comportamiento auditivo humano, cuya efectividad ha sido contrastada en tareas de reconocimiento del habla [51].

El procesador permite configurar el número de filtros, así como el rango de frecuencias en el que serán dispuestos.

Procesador `RASTA`

Aplica un filtrado RASTA (RelAtive SpecTrA) [78] a los parámetros de entrada. Aunque el filtro RASTA fue desarrollado originalmente para el procesamiento PLP

(Perceptual Linear Prediction) [77], es también utilizado con coeficientes cepstrales, bien a la salida del banco de filtros, bien directamente sobre los propios coeficientes [241]. Actúa como un filtro paso-banda que, por un lado, trata de eliminar variaciones lentas debidas al canal y, por otro lado, trata de suavizar variaciones rápidas posiblemente debidas a ruidos. En algunos casos, se ha podido comprobar que la normalización cepstral en tiempo real resulta más efectiva [79].

Procesador `DiscreteCosineTransform`

Calcula la transformada de coseno discreta (Discrete Cosine Transform, DCT) de cada vector de entrada. Aplicada al logaritmo de los coeficientes de un banco de filtros (la salida del Procesador `MelLogFilterBank`), la DCT obtiene, en sus primeros coeficientes, una representación decorrelada del tracto vocal (más concretamente la respuesta impulso del tracto vocal). Dichos coeficientes contienen información relativa al locutor, la articulación, etc. El Procesador `DiscreteCosineTransform` permite determinar el número de coeficientes a estimar, y ofrece la posibilidad de descartar el primer coeficiente (equivalente al logaritmo de la energía de la señal) y realzar los coeficientes centrales.

Procesador `Delta`

Obtiene una estimación de la primera y segunda derivada[8] de cada vector de entrada y devuelve a su salida un vector con la concatenación de los coeficientes estáticos y dinámicos (primeras y segundas derivadas). Este procesador no permite devolver un subconjunto de coeficientes (por ejemplo solo los estáticos y las primeras derivadas, o solo las primeras y segundas derivadas). No obstante, es posible añadir detrás el Procesador `VectorialDataFilter` (plugin de utilidades, Sección 4.1), el cual permite filtrar los campos deseados.

Procesador `ShiftedDelta`

Agrupa un conjunto de coeficientes dinámicos (solo primeras derivadas) en torno al instante actual. Aplicado a los coeficientes cepstrales, equivale a la denominada Shifted Delta Cepstrum (SDC) [94][178]. Los parámetros de salida contienen una mayor información dinámica, si bien ha de tenerse en cuenta que la dimensionalidad de los vectores de salida crece de forma significativa.

Procesador `MeanVarianceNormalization`

Elimina la componente continua de los parámetros de entrada, restando la media calculada sobre la señal completa, y opcionalmente, normaliza su varianza. Aplicado a los coeficientes cepstrales, equivale a la denominada Cepstral Mean Normalization (CMN) [21]. La normalización cepstral elimina de forma efectiva toda distorsión debida a un filtro estacionario, como pudieran serlo el tipo de micrófono, la distancia al micrófono o la acústica de la sala de grabación.

[8] A pesar de hacer uso del término derivada, realmente se trata de una estimación. Entiéndase que por primera y segunda derivada nos referimos a los coeficientes dinámicos de primer y segundo orden.

Procesador `LiveMeanNormalization`

Ofrece una alternativa en tiempo real al Procesador `MeanVarianceNormalization` (únicamente en lo referente a la normalización de la media). Se trata de un filtro pasa-alto que equivale a una normalización en la cual la media de los parámetros es actualizada dinámicamente.

Procesador `Gaussianization`

Aplica una transformación de gaussianización a los parámetros de entrada, de tal manera que los parámetros de salida sigan una distribución normal (Gaussiana) con media 0 y varianza 1. La transformación es estimada a partir de cada secuencia completa de datos de entrada. La gaussianización de una secuencia de parámetros es invariante a cualquier transformación monótona de los mismos, ya que el resultado de la transformación solo depende del rango de cada valor (su posición dentro del conjunto ordenado de valores).

Procesador `LiveGaussianization`

Aplica una transformación de gaussianización a los parámetros de entrada. A diferencia del Procesador `Gaussianization`, que estima la transformación a partir de la secuencia completa de parámetros, el Procesador `LiveGaussianization` realiza su estimación en base a una ventana deslizante. Este procesamiento (comúnmente denominado *short-time Gaussianization*) es utilizado en tareas de verificación del locutor, con ventanas del orden de tres segundos, incrementando la robustez frente a variabilidad de canal, ruido aditivo y, en cierta medida, frente a efectos no lineales debidos a variabilidad en los transductores (micrófonos) [119, 224].

Procesador `GLDSKernel`

Estima o aplica una expansión polinomial normalizada que, dada una secuencia de vectores de entrada, genera un único vector de salida de dimensión fija (independientemente de la longitud de la secuencia de entrada). El producto escalar de dichos vectores equivale a un kernel GLDS (Generalized Linear Discriminant Sequence Kernel)[44]. Este procesador requiere de una fase inicial en la que se estiman los parámetros de normalización a partir de todas las secuencias de datos de entrada (utilizando para ello un conjunto de entrenamiento). Finalizada la fase de estimación, es posible aplicar el procesador a cualquier otra secuencia de datos. El procesador da como salida un único vector de tamaño fijo para cada secuencia de vectores de entrada. Dicho vector contendrá los valores promediados y normalizados de las expansiones polinomial de los vectores de entrada. Cuando estos vectores son utilizados en una máquina de vectores soporte (SVM, Support Vector Machine) con una función de núcleo (kernel) lineal, dan lugar al denominado kernel GLDS. Los kernels GLDS han sido utilizados tanto en tareas de verificación del locutor como de verificación de la lengua [45].

4.5.2. Ejemplo de uso: Parametrización

A continuación se muestran tres ejemplos de parametrización de señales de audio para tres casos de uso: Reconocimiento Automático del Habla con señales de 16kHz, Reconocimiento del Locutor con señales de 8kHz y Reconocimiento de la Lengua con señales de 8kHz. En los tres casos, se entenderá que los recursos de audio están contenidos en una base de datos acústica y que la finalidad es generar un fichero de datos que contenga las señales parametrizadas.

La engine del Listado 4.4 muestra una parametrización típica para RAH de señales de 16kHz basada en la extracción de 13 cepstrales (MFCC, Mel-Frequency Cepstral Coefficients), sus primeras y segundas derivadas y la posterior normalización de la media cepstral. Como ya se ha comentado al inicio de la presente sección, los valores por defecto de los parámetros de la mayoría de los procesadores han sido establecidos tomando como referencia este caso de uso, por lo cual no es preciso modificar ninguno de sus valores, salvo los referidos a entrada y salida (base de datos y fichero de volcado) que son utilizados como parámetros de la engine. Nótese que antes de volcar la secuencia de datos a un fichero, se añade un chequeo de datos para detectar posibles errores numéricos. A continuación se muestra la página de manual de esta engine:

```
$ sautrela -help ASRParam16k.eng

ENGINE: ASRParam16k

  16kHz audio parametrization for Speech Recognition:

  13MFCC + Delta-DeltaDelta + CMN

   - 25ms windows, 10ms shift, HAMMING
   - PowerSpectrum (FFTsize=512)
   - MelLogFilterBank (40 filters, minFreq=130.0Hz, maxFreq=6800.0Hz)
   - DCT (energyIncluded, filters=13)
   - Deltas
   - Mean Normalization (CMN)

PARAMETERS:

  -d [URL,file:OPENDIALOG] the locator of the AcousticDataBase or "file:
    OPENDIALOG" for a File Open Dialog [databaseURL,ADBReader]
  -s [String,".*"] the sentences whose names match this regex are read [
    resourceNameRegex,ADBReader]
  -o [File,out.dump] the pathname of the dump file [streamFile,
    StreamWriter]
```

Nótese que la página de manual incluye una descripción detallada de los componentes de la engine. Esta descripción está contenida literalmente en el atributo `description` del descriptor XML de la engine.

La engine del Listado 4.5 muestra una parametrización apropiada para una tarea de Reconocimiento del Locutor con señales de 8kHz. El mero hecho de pasar de 16kHz a 8kHz hace necesaria la modificación de muchos parámetros, cuyos valores por defecto

Listado 4.4 Fichero `ASRParam16k.eng` de una engine de parametrización para el reconocimiento automático del habla.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE Engine SYSTEM "http://sautrela.org/lib/Engine.dtd">
<Engine name="ASRParam16k"
   description="16kHz audio parametrization for Speech Recognition:&#10;
   &#10;
   13MFCC + Delta-DeltaDelta + CMN&#10;
   &#10;
    - 25ms windows, 10ms shift, HAMMING&#10;
    - PowerSpectrum (FFTsize=512)&#10;
    - MelLogFilterBank (40 filters, minFreq=130.0Hz, maxFreq=6800.0Hz)&#10;
    - DCT (energyIncluded, filters=13)&#10;
    - Deltas&#10;
    - Mean Normalization (CMN)">
   <Processor name="ADBReader">
      <param name="databaseURL" value="?-d"/>
      <param name="resourceNameRegex" value="?-s"/>
   </Processor>
   <Buffer size="10"/>
   <Processor name="Windowing"/>
   <Buffer/>
   <Processor name="PowerSpectrum"/>
   <Buffer/>
   <Processor name="MelLogFilterBank"/>
   <Buffer/>
   <Processor name="DiscreteCosineTransform"/>
   <Buffer/>
   <Processor name="Delta"/>
   <Buffer/>
   <Processor name="MeanVarianceNormalization"/>
   <Buffer/>
   <Processor name="StreamTester"/>
   <Buffer/>
   <Processor name="StreamWriter" >
      <param name="streamFile" value="?-o"/>
   </Processor>
</Engine>
```

presuponen una frecuencia de muestreo de 16KHz. El ventaneo pasa a ser de 200 muestras y 80 muestras de desplazamiento (a pesar de tener la misma configuración temporal que la anterior engine: ventanas de 25ms y desplazamiento de 10ms). El modelado para el Reconocimiento del Locutor suele descartar las muestras (ventanas) que no contengan voz, por lo que se añade el procesador que estima una máscara de voz/no-voz, y al final de la engine se eliminan las muestras que no contenían voz. Dado que el tamaño de las ventanas es de 200 muestras, el espectro de potencia se calcula mediante una FFT de tamaño 256. El banco de filtros se ve modificado por la frecuencia de muestreo, reduciendo además el número de filtros a 20 (su valor por defecto es 24). El segundo factor que afecta al banco de filtros es el tipo de señal. A menudo, las señales de 8kHz han sido transmitidas mediante un canal telefónico. En tal caso, puede resultar interesante limitar el ancho de banda del banco de filtros al rango de $300 - 3300$ Hz. A la salida del banco de filtros se ha colocado un filtro

RASTA, para tratar de aliviar variabilidades debidas al canal. Por último, se aplica una gaussianización de los parámetros mediante una ventana de 300 muestras (3 segundos). Nótese que la gaussianización es aplicada una vez han sido eliminadas las muestras que no contienen voz. A continuación se muestra la página de manual de la engine del Listado 4.5:

```
$ sautrela -help SRParam8k.eng

ENGINE: SRParam8k

  8kHz audio parametrization for Speaker Recognition:

  VAD + 13MFCC + RASTA + Delta-DeltaDelta + Gaussianization

   - 25ms windows, 10ms shift, HAMMING
   - Energy based VAD (threshold=30dB)
   - PowerSpectrum (FFTsize=256)
   - MelLogFilterBank (20 filters, minFreq=300Hz, maxFreq=3300Hz)
   - RASTA
   - DCT (energyIncluded, filters=13)
   - Deltas
   - Short Time Gaussianization (windowSize=3s)

PARAMETERS:

  -d [URL,file:OPENDIALOG] the locator of the AcousticDataBase or "file:
    OPENDIALOG" for a File Open Dialog [databaseURL,ADBReader]
  -s [String,".*"] the sentences whose names match this regex are read [
    resourceNameRegex,ADBReader]
  -o [File,out.dump] the pathname of the dump file [streamFile,
    StreamWriter]
```

La engine del Listado 4.6 muestra una parametrización apropiada para una tarea de Reconocimiento de la Lengua con señales de 8kHz. Como puede observarse, no existen muchas diferencias frente a la parametrización diseñada para una tarea de Reconocimiento del Locutor (de hecho, ambas disciplinas han ido siempre de la mano). El tamaño de ventana es reducido a 160 muestras (20ms) y se estiman 7 cepstrales en vez de 13. La principal razón de reducir el número de cepstrales es que, en vez de calcular las primeras y segundas derivadas, suelen calcularse las denominadas *Shifted-Delta* (coeficientes dinámicos de primer orden en un conjunto de ventanas en torno a la ventana de análisis), concatenándolas junto a las características estáticas en un único vector, dando lugar a los parámetros SDC. El inconveniente de los parámetros SDC es que la concatenación de coeficientes dinámicos se realiza a costa de multiplicar por k (en nuestro caso, $k = 7$) la dimensión n de los parámetros de entrada, por lo que es recomendable partir de una dimensión reducida (de ahí que se usen solo $n = 7$ cepstrales). Una configuración 7-1-3-7 como la del ejemplo establece una dimensión de entrada de 7 (MFCCs) y la concatenación de 7 coeficientes dinámicos. Dado que además se incluyen los coeficientes estáticos, los vectores de salida son de dimensión

Listado 4.5 Fichero `SRParam8k.eng` de una engine de parametrización para reconocimiento de locutor.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE Engine SYSTEM "http://sautrela.org/lib/Engine.dtd">
<Engine name="SRParam8k"
   description="8kHz audio parametrization for Speaker Recognition:&#10;
   &#10;
   VAD + 13 MFCC + RASTA + Delta-DeltaDelta + Gaussianization&#10;
   &#10;
    - 25ms windows, 10ms shift, HAMMING&#10;
    - Energy based VAD (threshold=30dB)&#10;
    - PowerSpectrum (FFTsize=256)&#10;
    - MelLogFilterBank (20 filters, minFreq=300Hz, maxFreq=3300Hz)&#10;
    - RASTA&#10;
    - DCT (energyIncluded, filters=13)&#10;
    - Deltas &#10;
    - Short Time Gaussianization (windowSize=3s)">
   <Processor name="ADBReader">
      <param name="databaseURL" value="?-d"/>
      <param name="resourceNameRegex" value="?-s"/>
   </Processor> <Buffer size="10"/>
   <Processor name="Windowing">
      <param name="size" value="200"/>
      <param name="shift" value="80"/>
   </Processor> <Buffer/>
   <Processor name="VoiceActivityDetector">
      <param name="threshold" value="30"/>
   </Processor> <Buffer/>
   <Processor name="PowerSpectrum">
      <param name="size" value="256"/>
   </Processor> <Buffer/>
   <Processor name="MelLogFilterBank">
      <param name="filterNum" value="20"/>
      <param name="sampleRate" value="8000"/>
      <param name="minFreq" value="300.0"/>
      <param name="maxFreq" value="3300.0"/>
   </Processor> <Buffer/>
   <Processor name="RASTA"/> <Buffer/>
   <Processor name="DiscreteCosineTransform"/> <Buffer/>
   <Processor name="Delta"/> <Buffer/>
   <Processor name="UnvoicedFeatureRemover"/> <Buffer/>
   <Processor name="LiveGaussianization">
      <param name="windowSize" value="300"/>
   </Processor> <Buffer/>
   <Processor name="StreamTester"/> <Buffer/>
   <Processor name="StreamWriter" >
      <param name="streamFile" value="?-o"/>
   </Processor>
</Engine>
```

$7 + 7 \cdot 7 = 56$ (de haber mantenido los 13 MFCCs, los vectores serían de dimensión $13 + 13 \cdot 7 = 108$). Por último, se aplica una normalización de media y varianza a los parámetros SDC. A continuación se muestra la página de manual de la engine del Listado 4.6:

```
$ sautrela -help LRParam8k.eng

ENGINE: LRParam8k

  8kHz audio parametrization for Language Recognition:

  VAD + 7MFCC + RASTA + SDC 7-1-3-7 + MVN

   - 20ms windows, 10ms shift, HAMMING
   - Energy based VAD (threshold=30dB)
   - PowerSpectrum (FFTsize=256)
   - MelLogFilterBank (20 filters, minFreq=300Hz, maxFreq=3300Hz)
   - RASTA
   - DCT (energyIncluded, filters=7)
   - ShiftedDelta (SDC 7-1-3-7)
   - MVN

PARAMETERS:

  -d [URL,file:OPENDIALOG] the locator of the AcousticDataBase or "file:
    OPENDIALOG" for a File Open Dialog [databaseURL,ADBReader]
  -s [String,".*"] the sentences whose names match this regex are read [
    resourceNameRegex,ADBReader]
  -o [File,out.dump] the pathname of the dump file [streamFile,
    StreamWriter]
```

4.6. Plugin: Cuantificación Vectorial

El paquete `edu.gtts.sautrela.vq` contiene un conjunto básico de herramientas para estimar y aplicar una cuantificación vectorial a vectores de reales. La cuantificación vectorial simplifica el espacio de parámetros, sustituyendo cada vector por la etiqueta del centroide más próximo y permitiendo el uso de modelos basados en histogramas.

4.6.1. Procesadores

Procesador `CodebookTrainer`

Estima un conjunto de centroides (comúnmente llamado diccionario o codebook) a partir de las muestras de entrada. En una primera fase almacena todos los datos de

Listado 4.6 Fichero `LRParam8k.eng` de una engine de parametrización para reconocimiento de la lengua.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE Engine SYSTEM "http://sautrela.org/lib/Engine.dtd">
<Engine name="LRParam8k"
   description="8kHz audio parametrization for Language Recognition:&#10;
   &#10;
   VAD + 7 MFCC + RASTA + SDC 7-1-3-7 + MVN&#10;
   &#10;
    - 20ms windows, 10ms shift, HAMMING&#10;
    - Energy based VAD (threshold=30dB)&#10;
    - PowerSpectrum (FFTsize=256)&#10;
    - MelLogFilterBank (20 filters, minFreq=300Hz, maxFreq=3300Hz)&#10;
    - RASTA&#10;
    - DCT (energyIncluded, filters=7)&#10;
    - ShiftedDelta (SDC 7-1-3-7) &#10;
    - Mean and Variance Normalization">
   <Processor name="ADBReader">
      <param name="databaseURL" value="?-d"/>
      <param name="resourceNameRegex" value="?-s"/>
   </Processor> <Buffer size="10"/>
   <Processor name="Windowing">
      <param name="size" value="160"/>
      <param name="shift" value="80"/>
      <param name="function" value="HAMMING"/>
      <param name="cutDC" value="true"/>
   </Processor> <Buffer/>
   <Processor name="VoiceActivityDetector">
      <param name="threshold" value="30"/>
   </Processor> <Buffer/>
   <Processor name="PowerSpectrum">
      <param name="size" value="256"/>
   </Processor> <Buffer/>
   <Processor name="MelLogFilterBank">
      <param name="filterNum" value="20"/>
      <param name="sampleRate" value="8000"/>
      <param name="minFreq" value="300.0"/>
      <param name="maxFreq" value="3300.0"/>
   </Processor> <Buffer/>
   <Processor name="RASTA"/> <Buffer/>
   <Processor name="DiscreteCosineTransform">
      <param name="filterNum" value="7"/>
   </Processor> <Buffer/>
   <Processor name="ShiftedDelta">
      <param name="n" value="7"/>
      <param name="d" value="1"/>
      <param name="p" value="3"/>
      <param name="k" value="7"/>
      <param name="featurePrefixed" value="true"/>
   </Processor> <Buffer/>
   <Processor name="UnvoicedFeatureRemover"/> <Buffer/>
   <Processor name="MeanVarianceNormalization">
      <param name="varianceToOne" value="true"/>
   </Processor> <Buffer/>
   <Processor name="StreamTester"/> <Buffer/>
   <Processor name="StreamWriter" >
      <param name="streamFile" value="?-o"/>
   </Processor>
</Engine>
```

entrada en memoria[9] y a continuación trata de estimar el conjunto de centroides que minimice una cierta medida de la distorsión: la suma de los cuadrados de las distancias al centroide más próximo. El conjunto de centroides puede ser estimado mediante los algoritmos LGB [104] y eLBG [117], bien a partir de un conjunto vacío, bien a partir de un conjunto de centroides previamente estimado.

Procesador `Quantizer`

Carga una secuencia de vectores que representan al conjunto de centroides y realiza la cuantificación de los vectores de entrada. La salida del procesador es una secuencia de datos enteros de dimensión 1, cuyos valores corresponden a los índices del centroide más próximo a cada vector de entrada.

4.6.2. Ejemplo de uso: Cuantificación vectorial de muestras aleatorias

El Listado 4.7 muestra una engine que estima el codebook de cuantificación vectorial a partir de muestras aleatorias (vectores reales con valores en el rango [0,1]). El listado 4.8 muestra otra engine en la que se aplica una cuantificación vectorial a una secuencia de muestras aleatorias. A continuación se muestran las páginas de manual de ambas engines:

```
$ sautrela -help vqTrain.eng

ENGINE: vqTrain

  Train VQ from random data

PARAMETERS:

  -d [int,1] dimension of the vectors to generate [dataDim,
    RandomNumberGenerator]
  -s [int,5] number of streams to generate [streamNumber,
    RandomNumberGenerator]
  -l [int,10] number of vector per DataStream [streamLength,
    RandomNumberGenerator]
  -nc [int,10] number of centroids [cdbkSize,CodebookTrainer]
  -cdbk [File,out.cdbk] pathname for the trained CodeBook [finalCdbk,
    CodebookTrainer]
```

```
$ sautrela -help vqTest.eng

ENGINE: vqTest
```

[9] Dado que todos los datos de entrenamiento son cargados en memoria, el uso de cantidades elevadas de datos para estimar la cuantificación puede generar problemas de memoria.

Listado 4.7 `vqTrain.eng` - Descriptor XML de una engine de estimación de un codebook de cuantificación vectorial a partir de secuencias de vectores aleatorios.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!DOCTYPE Engine SYSTEM "http://sautrela.org/lib/Engine.dtd">
<Engine name="vqTrain" description="Train VQ from random data">
   <Processor name="RandomNumberGenerator">
      <param name="dataDim" value="?-d"/>
      <param name="streamNumber" value="?-s"/>
      <param name="streamLength" value="?-l"/>
   </Processor><Buffer/>
   <Processor name="CodebookTrainer">
      <param name="cdbkSize" value="?-nc"/>
      <param name="finalCdbk" value="?-cdbk"/>
   </Processor>
</Engine>
```

Listado 4.8 `vqTest.eng` - Descriptor XML de una engine de cuantificación vectorial de secuencias de vectores aleatorios.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!DOCTYPE Engine SYSTEM "http://sautrela.org/lib/Engine.dtd">
<Engine name="vqTest" description="Test VQ on random data">
   <Processor name="RandomNumberGenerator">
      <param name="dataDim" value="?-d"/>
      <param name="streamNumber" value="?-s"/>
      <param name="streamLength" value="?-l"/>
   </Processor><Buffer/>
   <Processor name="Quantizer">
      <param name="cdbkURL" value="?-cdbk"/>
   </Processor><Buffer/>
   <Processor name="Sniffer"/>
</Engine>
```

```
  Test VQ on random data

PARAMETERS:

  -d [int,1] dimension of the vectors to generate [dataDim,
    RandomNumberGenerator]
  -s [int,5] number of streams to generate [streamNumber,
    RandomNumberGenerator]
  -l [int,10] number of vector per DataStream [streamLength,
    RandomNumberGenerator]
  -cdbk [URL,file:OPENDIALOG] the locator of the CodeBook or "file:
    OPENDIALOG" for a File Open Dialog [cdbkURL,Quantizer]
```

En las siguientes líneas de comando se entrena primero un *codebook* de 8 centroides sobre un conjunto aleatorio de 10000 vectores bi-dimensionales, y posteriormente se realiza la cuantificación vectorial de dos secuencias aleatorias de longitud 4:

```
$ sautrela vqTrain.eng -d 2 -s 1 -l 10000 -nc 8 -cdbk my.cdbk
$ sautrela vqTest.eng -d 2 -s 2 -l 4 -cdbk my.cdbk
<StreamBegin msg="Random Double Data. Dim:2 Length:4"/>
<IntData value="6"/>
<IntData value="0"/>
<IntData value="0"/>
<IntData value="2"/>
<StreamEnd/>
<StreamBegin msg="Random Double Data. Dim:2 Length:4"/>
<IntData value="0"/>
<IntData value="1"/>
<IntData value="4"/>
<IntData value="7"/>
<StreamEnd/>
<EOS/>
```

4.7. Plugin: Modelado, entrenamiento y decodificación

Los paquetes `edu.gtts.sautrela.wfsa` y `edu.gtts.sautrela.wfsa.models` contienen un conjunto de herramientas de modelado basadas en autómatas de estados finitos ponderados (WFSA, por sus siglas en inglés).

Sautrela cuenta con un conjunto de modelos que da cobertura al modelado acústico, léxico y del lenguaje. Entre los modelos implementados podemos encontrar sencillos autómatas de estados finitos deterministas y no deterministas, modelos ocultos de Markov discretos y continuos, y modelos k-explorables en sentido estricto (n-gramas) [174, 26, 40, 179]. Además implementa los modelos de Markov por capas (LMM, *Layered Markov Models*), los cuales permiten integrar en un solo autómata o modelo diversos niveles de conocimiento [121]. Cada uno de los modelos implementados cuenta además con su correspondiente comando, que permitirá crear o manipular el modelo en cuestión.

El módulo de modelado contiene, entre otros, dos procesadores fundamentales: el decodificador y el entrenador. Ambos pueden funcionar frente a cualquiera de los modelos previamente citados. Como se mostrará más adelante, los modelos implementan una interfaz de decodificación que, en combinación con parámetros de poda, es utilizada para obtener la decodificación más probable dada una entrada de datos. De esta manera, un único procesador sirve tanto para crear un sencillo decodificador acústico-fonético, un reconocedor de comandos de voz, o un complejo reconocedor de habla continua. El decodificador, basándose únicamente en la interfaz de decodificación, realiza la búsqueda de la secuencia de transiciones que maximice la verosimilitud. Dado que solo requiere acceso a la interfaz de decodificación, el decodificador puede ignorar por completo la naturaleza interna del modelo utilizado. Este desacople entre la naturaleza interna del modelo y su uso a la hora de decodificar una secuencia de datos de entrada permite a Sautrela definir un conjunto extensible de modelos: todo

modelo (sea propio de Sautrela o creado por terceros) que implemente la interfaz de decodificación es directamente integrable en el sistema, pudiendo ser utilizado por el procesador decodificador. De forma análoga, los modelos implementan una interfaz de entrenamiento que, bajo un esquema de aprendizaje de esperanza-maximización (EM) permite desacoplar la fase de estimación de la esperanza de la verosimilitud (tarea realizada por el entrenador) de la fase de maximización u optimización de parámetros (tarea realizada por el modelo) [124, 126]. Un único procesador de entrenamiento se encarga de realizar las tareas de entrenamiento independientemente de cuál sea la naturaleza interna del modelo (o modelos) en cuestión. De forma análoga a lo descrito en referencia a la decodificación, todo modelo (sea propio de Sautrela o creado por terceros) que implemente la interfaz de entrenamiento es directamente integrable en el sistema, y podrá ser estimado (o re-estimado) mediante el procesador entrenador. El Capítulo 5 desarrolla más en profundidad las aportaciones de Sautrela en lo referente al uso de interfaces de decodificación y entrenamiento, conceptos ambos que han sido denominados como decodificación unificada (Sección 5.1) y entrenamiento unificado (Sección 5.2), respectivamente.

A continuación se mostrará la jerarquía de modelos de Sautrela, el mecanismo de instanciación y volcado de modelos, así como la descripción de cada uno de los modelos implementados. Se continuará con los comandos y procesadores incorporados en el plugin de modelado, y se finalizará con varios ejemplos de uso.

4.7.1. Jerarquía de modelos

Sautrela define una jerarquía de modelos basada en WFSAs con las siguientes características:

- Todo autómata tiene un único estado inicial.
- Las emisiones ocurren en las transiciones.
- Una transición queda definida por un estado origen, un símbolo de emisión y un estado destino. Además, toda transición lleva asociada una probabilidad.
- Todo estado puede tener una probabilidad no nula de ser final. La suma de dicha probabilidad final y las probabilidades del conjunto de transiciones que parten de él es la unidad.
- Un WFSA es determinista si para todo estado origen y símbolo[10] de emisión existe a lo sumo una única transición. De lo contrario, es no-determinista.

Para ofrecer la máxima flexibilidad a la hora de implementar un WFSA, los estados, símbolos y transiciones son declarados como interfaces. El Listado 4.9 muestra las tres interfaces en cuestión[11]. Como puede verse, `State` y `Symbol` son sencillas interfaces

[10] En la presente memoria, el término símbolo se refiere a una observación del modelo implementado por un WFSA. Puede representar tanto observaciones pertenecientes a un alfabeto finito, como observaciones pertenecientes a un alfabeto infinito (que puede o no representar un espacio continuo). A su vez, puede estar compuesto por un único valor o múltiples valores, así como ser de naturaleza textual o numérica, etc. El uso de interfaces permite a Sautrela interaccionar con todo tipo de símbolo sin necesidad de conocer su naturaleza (implementación).

[11] En la presente memoria, y para facilitar la lectura, se ha evitado la notación de Genéricos en Java. La documentación completa del código fuente puede ser consultada en `http://www.sautrela.org`.

Listado 4.9 Interfaces State, Symbol, Transition y Alphabet.

```
public interface Named {
   public String getName();
}

public interface State {}

public interface Symbol extends Named {}

public interface Transition extends Named,Comparable {
   public State getSource();
   public State getDestination();
   public Symbol getSymbol();
   public double getProbability();
}

public interface Alphabet extends Iterable {
   public Symbol valueOf(String name);
}
```

que permiten la abstracción de las clases subyacentes (de hecho, solo se establece que deben poseer un atributo de nombre), mientras que la interfaz `Transition` declara un conjunto de métodos que permiten acceder a sus tres elementos constituyentes (estado origen, símbolo de emisión y estado destino) así como la probabilidad asociada. Además de las tres interfaces anteriores, se define también la interfaz `Alphabet`, que representa un alfabeto de símbolos asociado a un WFSA. Como se verá en la Sección 5.1, esta interfaz contiene el método que sirve de nexo de unión entre las secuencias de observaciones a procesar y el WFSA en cuestión.

El uso de interfaces ofrece un grado de abstracción que permite dar cobertura a un gran número de diferentes tipos de modelos. Por ejemplo, un modelo léxico que represente las posibles pronunciaciones de una palabra tiene un alfabeto de emisión discreto (el conjunto de unidades fonéticas), mientras que un modelo acústico que represente una unidad fonética puede definir sus emisiones en un espacio vectorial continuo (el espacio de parámetros acústicos). La abstracción mediante interfaces permite trabajar indistintamente con emisiones en un espacio discreto o continuo, escalar o vectorial.

A partir de las interfaces previamente presentadas, se ha definido la interfaz `WFSA` (véase Listado 4.10) que, extendida por sus dos variantes, determinista y no-determinista, define el conjunto de métodos que todo WFSA debe implementar para que sea posible integrarlo en Sautrela. La interfaz puede dividirse en tres partes: la primera contiene los elementos que permiten la instanciación y volcado de los autómatas, la segunda (en conjunción con los métodos declarados en las dos sub-interfaces) define las funcionalidades relativas a la decodificación, y la última reúne los métodos asociados al entrenamiento. En la presente sección se describe únicamente la primera de las partes, ya que los métodos relativos a la decodificación y el entrenamiento serán abordados en las Secciones 5.1 y 5.2 del siguiente capítulo.

Listado 4.10 Interfaz de un WFSA, junto a sus dos sub-interfaces: determinista (DWFSA) y no-determinista (NdWFSA).

```
public interface WFSA extends Named {
   /** Instanciación y volcado **/
   public interface Factory {
      public WFSA getInstance(InputSource is);
      public WFSA getInstance(InputSource is,Alphabet a);
   }
   public String toXML();

   /** Decodificación **/
   public Alphabet getAlphabet();
   public State getIniState();
   public double getFinProb(State s);
   public Transition[] getTrans(State s);
   public T getRandomTrans(S state);

   /** Entrenamiento **/
   public void priorExpectation(double expectation);
   public void addExpectation(Transition t,double expectation);
   public void addFinalExpectation(State s,double expectation);
   public void dumpSuffStats();
}

public interface DWFSA extends WFSA {
   public Transition getTrans(State s,Symbol y);
}

public interface NdWFSA extends WFSA {
   public Transition[] getTrans(State s,Symbol y);
}
```

4.7.2. Instanciación y volcado de WFSAs

De manera análoga a los procesadores presentados en el capítulo anterior, Sautrela establece un patrón de diseño que permite integrar WFSAs propios así como WFSAs desarrollados por terceros. Para ello, se establece que todo WFSA debe ser instanciable a partir de un descriptor XML como el que se muestra en el Listado 4.11. Dicho XML debe contener toda la información necesaria para instanciar el autómata, si bien la estructura interna del descriptor dependerá de cada implementación. Solo se exige que el elemento `WFSA` contenga al menos un atributo `className` que indique el nombre de la clase a instanciar.

Además, se establece que todo autómata debe contener un campo estático[12] que implemente la interfaz `WFSA.Factory` del Listado 4.10. Esta interfaz contiene los dos

[12] Al tratarse de un campo estático, es posible acceder a él directamente a través del nombre de clase. El desarrollador es libre de escoger el nombre de dicho campo, ya que este es localizado dinámicamente mediante la introspección [13] en el momento de instanciar el autómata.

Listado 4.11 Descriptor XML de un WFSA genérico. Para que un WFSA pueda ser instanciado por Sautrela, el elemento `WFSA` de su descriptor XML debe contener un atributo `className` que indique el nombre completo de la clase responsable de instanciar el autómata.

```
<?xml version="1.0" encoding="UTF-8" ?>
<WFSA className="the.class.name" ... >
   ...
   ...
</WFSA>
```

Listado 4.12 Ejemplo de implementación de la interfaz `WFSA.Factory`. La clase responsable de instanciar un modelo WFSA debe contener un campo estático que implemente la interfaz `WFSA.Factory` (véase Listado 4.10).

```
public class MiWFSA implements WFSA {
   ...
   public static WFSA.Factory miFactoria = new WFSA.Factory(){
      public WFSA getInstance(InputSource is){
         ...
      }
      public WFSA getInstance(InputSource is,Alphabet alphabet){
         ...
      }
   };
   ...
}
```

Listado 4.13 Descriptor XML de un conjunto de WFSAs. Un conjunto de WFSAs está compuesto por una colección de WFSAs con nombres distintos y que comparten un mismo alfabeto. Los WFSAs pueden corresponder a clases diferentes.

```
<?xml version="1.0" encoding="UTF-8" ?>
<WFSASet name="theName">
   <WFSA className="the.class.name1" ... >
      ...
      ...
   </WFSA>
   <WFSA className="the.class.name2" ... >
      ...
      ...
   </WFSA>
   ...
   ...
</WFSASet>
```

métodos que hacen posible la instanciación del autómata a partir de un descriptor XML. El segundo de los métodos de la interfaz, el que contiene un alfabeto como segundo argumento, permite instanciar un WFSA haciendo uso de un alfabeto ya existente, lo cual hace posible instanciar un conjunto de autómatas que compartan un único alfabeto de símbolos de emisión. La instanciación de un WFSA a partir de un XML consta de las siguientes fases:

1. Se inspecciona el XML en busca del nombre de la clase a instanciar.
2. Se localiza en dicha clase un campo estático que implemente la sub-interfaz `WFSA.Factory`.
3. Se invoca el método `getInstance`, pasándole el propio XML como argumento.

En contraposición al presente patrón de diseño, resultaría igualmente válido (y en cierta manera más sencillo) establecer que todo WFSA tuviera dos constructores equivalentes a los dos métodos de la interfaz `WFSA.Factory`. Sin embargo, ello plantearía ciertos problemas. Por un lado, es imposible forzar la implementación de constructores en Java, de tal manera que al igual que ocurre con el patrón de diseño propuesto, la comprobación debería realizarse en tiempo de ejecución. Por otro lado, y más importante aún, durante el desarrollo no es posible comprobar de manera automática si la implementación ha sido acorde al patrón de diseño. Por el contrario, la declaración de un campo estático resulta extremadamente sencilla y, una vez realizada, fuerza al desarrollador a implementar correctamente los métodos instanciadores[13]. El código del Listado 4.12 muestra un ejemplo de implementación de un WFSA en el que se define un campo estático que implementa la sub-interfaz `WFSA.Factory`.

El volcado de los modelos se realiza a partir del XML que devuelve el método `toXML` incluido en la interfaz. Como ya se comentó en la introducción del presente capítulo, el uso del XML para la representación de autómatas evitará los problemas derivados de la coexistencia de diferentes codificaciones de texto[14].

Los modelos pueden agruparse en conjuntos de modelos. Un conjunto de modelos contiene una colección de modelos con nombres distintos y que comparten un mismo alfabeto. Un conjunto de modelos puede construirse mediante el comando `WFSASet` (véase Subsección 4.7.5) El Listado 4.13 muestra un ejemplo de formato de un descriptor XML de un conjunto de WFSAs.

4.7.3. Modelos implementados

La Figura 4.3 muestra el conjunto de modelos incluidos en Sautrela. Todos ellos implementan bien la interfaz determinista `DWFSA`, bien la no-determinista `NdWFSA`, y cuentan con comandos para crearlos y manipularlos. A continuación se presenta una breve descripción de cada uno de ellos.

[13] En realidad, el patrón de diseño más sencillo se basaría en declarar en la interfaz `WFSA` un método estático `public static Factory getFactory();` que forzase a implementar un método que retorna una factoría de modelos. Sin embargo, Java no ha permitido la declaración de métodos estáticos en las interfaces hasta la reciente versión `1.8`, por lo que en su día se optó por esta segunda opción.

[14] Dado que el método `toXML` devuelve una cadena de caracteres que contiene el XML, utiliza la codificación Unicode, y por tanto no precisa generar cabecera alguna, que será descartada por Sautrela.

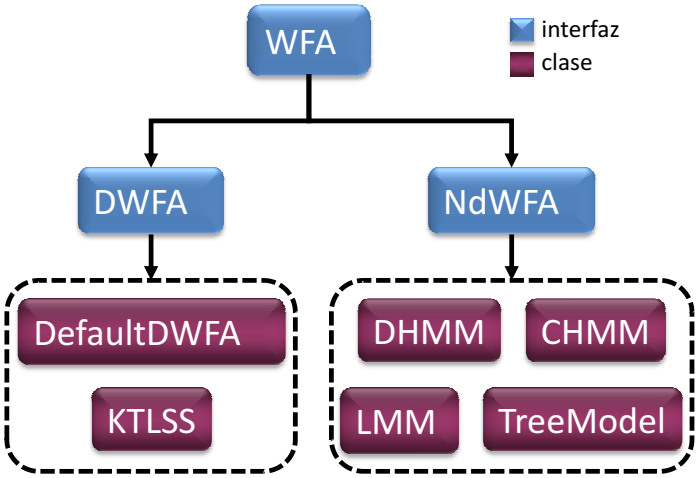


Figura 4.3 Conjunto de modelos implementados en Sautrela.

DefaultDWFSA (*Deterministic Weighted Finite-State Automata*)

Se trata de la implementación por defecto de un WFSA determinista (dado un estado origen y un símbolo del alfabeto, existe a lo sumo una única transición posible desde ese estado y con ese símbolo), que define de forma explícita todos sus estados y transiciones. Este autómata es adecuado para representar modelos léxicos y gramáticas sencillas, y es también utilizado para representar la capa de supervisión (la transcripción de las señales de entrada) durante el entrenamiento de un conjunto de modelos. La Figura 4.4 muestra un WFSA determinista que modela la pronunciación de la palabra *"descripción"*, contemplando la posibilidad de que los fonemas `p` y `n` no sean pronunciados (es posible transitar desde el estado 6 al estado 8, evitando así el símbolo `p`, y el estado 10 es final, con lo que puede evitarse el símbolo `n` final). El Listado 4.14, por su parte, muestra el descriptor XML de un modelo `DefaultDWFSA` que implementa el WFSA de la Figura 4.4.

KTLSS (*K-Testable Language in the Strict Sense*)

Un modelo k-testable en sentido estricto (KTLSS, por sus siglas en inglés) [174, 26, 40, 179] es equivalente a los modelos de N-gramas [89]. En Sautrela, un KTLSS implementa la interfaz WFSA determinista y suele ser utilizado como modelo de lenguaje. No existe limitación alguna ni en el valor K (equivalente al valor N de los N-gramas) ni en el tamaño del vocabulario, y ofrece cobertura a los símbolos fuera de vocabulario. A diferencia del resto de modelos, los KTLSS ofrecen la posibilidad de modificar su topología durante el entrenamiento, resultando posible partir de un modelo "vacío" o de un modelo previamente entrenado.

La cabecera es generada por Sautrela al volcar el XML al dispositivo de salida, momento en el que se define la codificación utilizada a tal efecto.

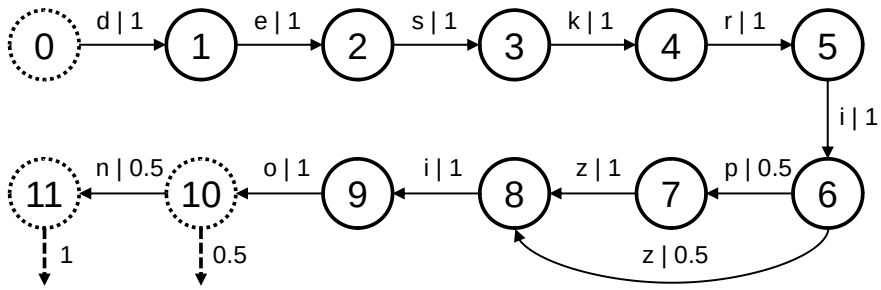


Figura 4.4 Modelo de pronunciación de la palabra *"descripción"* mediante un WFSA determinista. El modelo de pronunciación contempla la posibilidad de que los fonemas `p` y `n` no sean pronunciados.

Listado 4.14 Descriptor XML de un modelo `DefaultDWFSA` que representa la pronunciación de la palabra *"descripción"*. Las probabilidades pueden ser dadas en base a una escala lineal (atributo `linProb`) o en base a una escala logarítmica natural (atributo `logProb`).

```
<?xml version="1.0" encoding="UTF-8" ?>
<WFSA name="descripción" className="edu.gtts.sautrela.wfsa.models.DefaultDWFSA">
   <IniState name="0"/>
   <FinState name="10" linProb="0.5"/>
   <FinState name="11" linProb="1.0"/>
   <Transition from="0" to="1" symbol="d" linProb="1.0"/>
   <Transition from="1" to="2" symbol="e" linProb="1.0"/>
   <Transition from="2" to="3" symbol="s" linProb="1.0"/>
   <Transition from="3" to="4" symbol="k" linProb="1.0"/>
   <Transition from="4" to="5" symbol="r" linProb="1.0"/>
   <Transition from="5" to="6" symbol="i" linProb="1.0"/>
   <Transition from="6" to="7" symbol="p" linProb="0.5"/>
   <Transition from="6" to="8" symbol="z" linProb="0.5"/>
   <Transition from="7" to="8" symbol="z" linProb="1.0"/>
   <Transition from="8" to="9" symbol="i" linProb="1.0"/>
   <Transition from="9" to="10" symbol="o" linProb="1.0"/>
   <Transition from="10" to="11" symbol="n" linProb="0.5"/>
</WFSA>
```

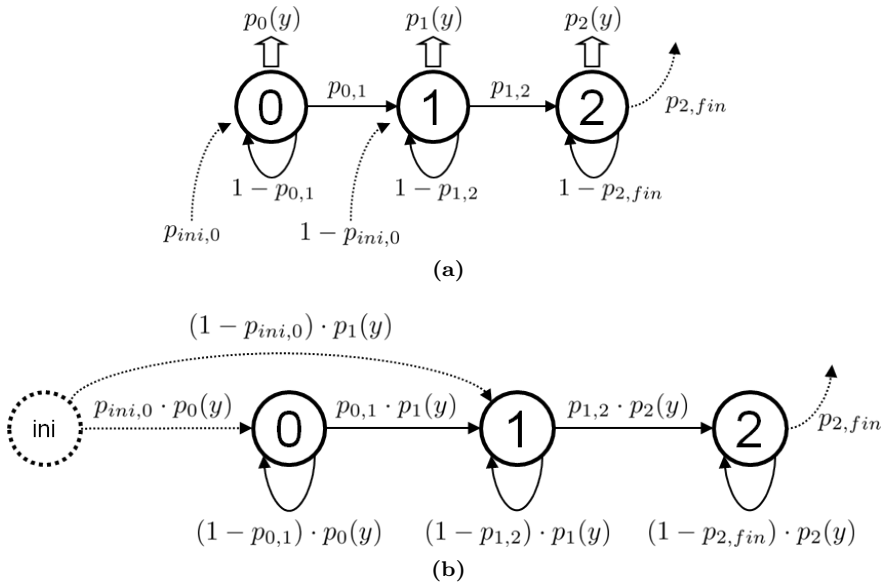


Figura 4.5 Comúnmente, un HMM (a) genera emisiones en los estados y puede tener más de un estado inicial, mientras que su WFSA equivalente (b) funde las emisiones con las transiciones. El autómata equivalente posee un estado adicional que es además el inicial. Este estado inicial puede transitar a todos los estados iniciales del HMM.

DHMM (*Discrete Hidden Markov Model*)

Representa un modelo oculto de Markov (HMM, por sus siglas en inglés) con emisiones discretas e implementa la interfaz WFSA no-determinista. No existe limitación alguna en el tamaño del modelo (número de estados) y su topología.

Contrariamente a lo dispuesto para los WFSAs, en un HMM las emisiones ocurren en los estados, y no en las transiciones. Por ello, la implementación de la interfaz WFSA se realiza mediante un autómata equivalente que combina las transiciones y emisiones: cada transición del HMM subyacente da lugar en el modelo equivalente a tantas transiciones como emisiones existen en el estado destino[15]. Además, y dado que los estados iniciales de un HMM deben generar una emisión, el autómata equivalente contiene un estado inicial adicional del cual parten transiciones a todos los estados iniciales del HMM. La Figura 4.5 muestra un HMM y su WFSA equivalente.

El Listado 4.15 muestra el ejemplo de un descriptor XML de un HMM discreto como el de la Figura 4.5. El modelo tiene una topología lineal con bucles (un estado puede transitar sobre sí mismo o al siguiente estado) compuesta por tres estados, dos de ellos iniciales y un único estado final. El alfabeto de emisión está compuesto únicamente por dos símbolos: $\{A, B\}$.

[15]Este planteamiento equivale a suponer que las emisiones del HMM ocurren justo antes de llegar al estado destino.

Listado 4.15 Descriptor XML de un modelo `HMM`. El modelo tiene una topología lineal con bucles compuesta por tres estados, con dos estados iniciales y un único estado final. El alfabeto de emisión está compuesto únicamente por dos símbolos: $\{A, B\}$.

```
<?xml version="1.0" encoding="UTF-8" ?>
<WFSA name="myHMM" size="3" cdbkSize="2" className="edu.gtts.sautrela.wfsa.models.
    HMM">
  <IniState name="0" linProb="0.6"/>
  <IniState name="1" linProb="0.4"/>
  <FinState name="2" linProb="0.2"/>
  <Transition from="0" to="0" linProb="0.7"/>
  <Transition from="0" to="1" linProb="0.3"/>
  <Transition from="1" to="1" linProb="0.9"/>
  <Transition from="1" to="2" linProb="0.5"/>
  <Transition from="2" to="2" linProb="0.8"/>
  <Emission state="0" symbol="A" linProb="0.5"/>
  <Emission state="0" symbol="B" linProb="0.5"/>
  <Emission state="1" symbol="A" linProb="0.3333333333"/>
  <Emission state="1" symbol="B" linProb="0.6666666667"/>
  <Emission state="2" symbol="A" linProb="0.25"/>
  <Emission state="2" symbol="B" linProb="0.75"/>
</WFSA>
```

CHMM (*Continuous Hidden Markov Model*)

Representa un HMM continuo que implementa la interfaz WFSA no-determinista. Estos modelos son comúnmente utilizados para el modelado de unidades acústicas (fonemas, difonemas, trifonemas, etc.). Análogamente a lo descrito para los HMM discretos, el caso continuo también requiere de la generación dinámica de un WFSA equivalente. Además, al tratarse de un autómata con emisiones en un espacio continuo (no discreto), las transiciones son generadas dinámicamente (existen infinitas transiciones que parten de un mismo estado) bajo demanda. Por último, el alfabeto asociado a un HMM continuo es intrínsecamente infinito.

LMM (*Layered Markov Model*)

Sautrela es el entorno en el que se formulan por vez primera los modelos de Markov por capas o LMMs [121]. Un LMM integra en un único WFSA no-determinista distintos niveles de conocimiento representados por conjuntos de autómatas. Los modelos por capas son una pieza clave en la arquitectura de modelos de Sautrela, ya que su formalismo ofrece una gran flexibilidad a la hora de definir escenarios tanto de reconocimiento como de aprendizaje.

Un LMM consta de diversas capas, cada una correspondiente a una fuente o nivel de conocimiento. Tomando como ejemplo el caso de un reconocedor del habla continua, podríamos distinguir tres capas de conocimiento: las referidas al modelado acústico, el modelado léxico y el modelado de lenguaje. Cada capa, formada por un conjunto de WFSAs (salvo la última, que constará siempre de un único WFSA), modela sus unidades en términos de las unidades pertenecientes a la capa directamente inferior. Así, y siguiendo con el ejemplo anterior, los modelos léxicos definen las posibles combinaciones de fonemas que dan lugar a una palabra, mientras que el modelo de lenguaje

define las posibles combinaciones de palabras que dan lugar a una frase[16].

Desde el punto de vista de la decodificación, un LMM es equivalente al resultado de la composición de transductores, en la que cada uno de los transductores representa una capa de conocimiento. Sin embargo, y a diferencia de los entornos o *toolkits* relacionados con las tecnologías del habla y que hacen uso de transductores a la hora de la decodificación (integrando todas las capas de conocimiento en un único transductor compuesto), un LMM tiene la capacidad de ser entrenado, transfiriendo dicho entrenamiento a cada uno de los modelos que lo compone. El uso de LMMs como herramienta de entrenamiento de modelos ofrece un nivel de abstracción que permite definir de manera sencilla infinidad de esquemas diferentes de entrenamiento de un conjunto de modelos. La Sección 5.4 contiene una descripción más extensa de los LMMs.

TreeModel

Un TreeModel es un WFSA no-determinista que integra en un solo autómata un léxico en árbol y un modelo de lenguaje, ofreciendo una decodificación basada en árbol léxico. El modelo de lenguaje puede estar representado por cualquier WFSA determinista, y el vocabulario puede contener transcripciones múltiples. Un TreeModel resulta de especial interés cuando se trabaja con grandes vocabularios, ya que reduce drásticamente el espacio de búsqueda [113] y las consultas al modelo de lenguaje. La Sección 5.7 ofrece una descripción más detallada de este modelo.

4.7.4. Integración de nuevos modelos

Como ya se ha indicado anteriormente, Sautrela define un conjunto extensible de modelos. El uso de interfaces a la hora de definir abstracciones funcionales ofrece la posibilidad de integrar tanto modelos propios como de terceros. Sautrela no establece (de hecho ignora) ni la estructura del descriptor XML (más allá de la existencia de un elemento raíz `WFSA` que contenga el atributo `className`), ni la naturaleza interna de los modelos (más allá de la exigencia de que implementen la interfaz `WFSA`, en alguna de sus dos variantes `DWFSA` o `NdWFSA`, y que contengan a su vez un campo estático que implemente la interfaz `WFSA.Factory`). Todo modelo cuya clase cumpla con estos requisitos y esté contenida en un plugin accesible podrá ser integrado en Sautrela, pudiendo ser objeto de las cuatro operaciones básicas:

1. Instanciación a partir de un descriptor XML.

2. Decodificación de una secuencia de observaciones de entrada, generando la correspondiente secuencia óptima de transiciones.

3. Entrenamiento del modelo a partir de un conjunto de secuencias de observaciones de entrada.

4. Volcado a un descriptor XML.

[16] Tal vez un término más correcto sería modelo de frase, ya que los llamados modelos de lenguaje suelen modelar las frases, no el lenguaje en sí mismo.

A diferencia de los procesadores y los comandos, dado que los modelos son identificados por su nombre completo de clase, no es necesario añadir ninguna entrada al manifiesto del plugin.

4.7.5. Comandos

Comando `WFSASet`

Crea un conjunto de WFSAs a partir de sus localizadores. Los componentes del conjunto deben tener nombres diferentes (el nombre se obtiene mediante el método `getName()` de la interfaz `WFSA`) y deben compartir un mismo alfabeto. En principio, si bien es extraño que se dé la situación, es posible crear un conjunto de WFSAs a partir de modelos de distintos tipos, siempre que estos puedan compartir un mismo alfabeto. Existen comandos específicos (`DefaultDWFSASet` y `CHMMSet`) que permiten crear o manipular conjuntos de modelos de un tipo concreto.

Comando `DefaultDWFSA`

Crea un WFSA determinista a partir de una transcripción, con posibilidad de permitir la inserción de símbolos (típicamente símbolos de relleno o silencio) al inicio, al final y entre cada una de las unidades.

La siguiente línea de comando, por ejemplo, crea un WFSA determinista que representa una posible locución de la palabra `"casa"`:

```
$ sautrela DefaultDWFSA -n casa k a s a
<?xml version="1.0" encoding="UTF-8" ?>
<WFSA name="casa" className="edu.gtts.sautrela.wfsa.models.DefaultDWFSA">
  <IniState name="0"/>
  <FinState name="4" linProb="1.0"/>
  <Transition from="0" to="1" symbol="k" linProb="1.0"/>
  <Transition from="1" to="2" symbol="a" linProb="1.0"/>
  <Transition from="2" to="3" symbol="s" linProb="1.0"/>
  <Transition from="3" to="4" symbol="a" linProb="1.0"/>
</WFSA>
```

El siguiente ejemplo crea un WFSA determinista que representa la secuencia de palabras que da lugar a un comando de voz. Dado que el audio podría contener silencios al inicio, al final y entre las palabras que forman el comando de voz, se permiten silencios en todas partes:

```
$ sautrela DefaultDWFSA -n comando1 -lris SIL iniciar navegador web
<?xml version="1.0" encoding="UTF-8" ?>
<WFSA name="comando1" className="edu.gtts.sautrela.wfsa.models.
    DefaultDWFSA">
  <IniState name="0"/>
  <FinState name="3" linProb="0.5"/>
  <FinState name="i3" linProb="1.0"/>
  <Transition from="0" to="i0" symbol="SIL" linProb="0.5"/>
  <Transition from="0" to="1" symbol="iniciar" linProb="0.5"/>
```

```
  <Transition from="1" to="i1" symbol="SIL" linProb="0.5"/>
  <Transition from="1" to="2" symbol="navegador" linProb="0.5"/>
  <Transition from="2" to="i2" symbol="SIL" linProb="0.5"/>
  <Transition from="2" to="3" symbol="web" linProb="0.5"/>
  <Transition from="3" to="i3" symbol="SIL" linProb="0.5"/>
  <Transition from="i0" to="1" symbol="iniciar" linProb="1.0"/>
  <Transition from="i1" to="2" symbol="navegador" linProb="1.0"/>
  <Transition from="i2" to="3" symbol="web" linProb="1.0"/>
</WFSA>
```

Comando `Dictionary`

Crea un diccionario de transcripciones a partir de un recurso de texto que contenga las transcripciones. La ventaja de usar un diccionario frente a un fichero de texto reside principalmente en que se eliminan los problemas de codificación, ya que el diccionario resultante es volcado en forma de documento XML. Soporta además transcripciones múltiples, es decir, un mismo término puede contener más de una transcripción. El siguiente ejemplo muestra la creación de un sencillo diccionario que contiene transcripciones de tres palabras:

```
$ sautrela Dictionary /dev/stdin << EOF
CASA k a s a
SALUD s a l u d
CAZA k a z a
SALUD s a l u z
EOF
<?xml version="1.0" encoding="UTF-8" ?>
<Dictionary>
  <Key name="CASA">
    <Value value="k a s a"/>
  </Key>
  <Key name="CAZA">
    <Value value="k a z a"/>
  </Key>
  <Key name="SALUD">
    <Value value="s a l u d"/>
    <Value value="s a l u z"/>
  </Key>
</Dictionary>
```

Comando `DefaultDWFSASet`

Crea un conjunto de WFSAs deterministas a partir de un diccionario. Genera para cada una de las entradas del diccionario su correspondiente modelo `DefaultDWFSA`. No soporta diccionarios que contengan transcripciones múltiples.

Comando DHMM

Crea un HMM discreto con el nombre y número de estados indicado, usando bien una topología lineal con bucles (el único estado inicial es el primero y el único estado final es el último), o bien una topología ergódica (todos los estados son iniciales y finales, pudiendo transitar entre cualesquiera dos estados). El alfabeto de emisión puede ser suministrado bien mediante un rango de números enteros, o bien explícitamente. Todas las probabilidades de emisión y las correspondientes a la topología (probabilidades iniciales, finales y de transición) son inicializadas de acuerdo a una distribución uniforme, salvo en el caso de la topología ergódica, en la cual se añade una pequeña perturbación a dichas probabilidades para romper la simetría del modelo[17]. A continuación se muestra la página de manual del comando:

```
$ sautrela -help DHMM

  DHMM (edu.gtts.sautrela.wfsa.models.DHMM)

    create a discrete Hidden Markov Model

  Syntax:  DHMM  [-E] [-n name] [-r from,to] [-s symbol[,symbol ...]] size

    -E  Use an ergodic topology. Default topology is linear with loops.
    -n name  The name of the DHMM
    -r from,to  Integer range of emission symbols
    -s symbol[,symbol ...]  List of emission symbols
    size  Number of states
```

La siguiente línea de código genera un HMM lineal de tres estados con un alfabeto de emisión $A = \{1, 2, 3, 4\}$:

```
$ sautrela DHMM -n myHMM -r 1,4 3
<?xml version="1.0" encoding="UTF-8" ?>
<WFSA name="myHMM" size="3" cdbkSize="4" className="edu.gtts.sautrela.wfsa
    .models.DHMM">
  <IniState name="0" linProb="1.0"/>
  <FinState name="2" linProb="1.0"/>
  <Transition from="0" to="0" linProb="0.5"/>
  <Transition from="0" to="1" linProb="0.5"/>
  <Transition from="1" to="1" linProb="0.5"/>
  <Transition from="1" to="2" linProb="0.5"/>
  <Transition from="2" to="2" linProb="0.5"/>
  <Emission state="0" symbol="1" linProb="0.25"/>
  <Emission state="0" symbol="2" linProb="0.25"/>
  <Emission state="0" symbol="3" linProb="0.25"/>
  <Emission state="0" symbol="4" linProb="0.25"/>
  <Emission state="1" symbol="1" linProb="0.25"/>
```

[17] Un modelo ergódico simétrico es equivalente a otro modelo con un único estado. La simetría conlleva la no especialización de los estados (las re-estimaciones derivadas del entrenamiento son idénticas para cada uno de ellos).

```
  <Emission state="1" symbol="2" linProb="0.25"/>
  <Emission state="1" symbol="3" linProb="0.25"/>
  <Emission state="1" symbol="4" linProb="0.25"/>
  <Emission state="2" symbol="1" linProb="0.25"/>
  <Emission state="2" symbol="2" linProb="0.25"/>
  <Emission state="2" symbol="3" linProb="0.25"/>
  <Emission state="2" symbol="4" linProb="0.25"/>
</WFSA>
```

En el siguiente ejemplo se crea un modelo ergódico de tres estados y alfabeto $A = \{X, Y\}$. Nótese que todas las probabilidades relativas a la topología sufren una pequeña perturbación que elimina la simetría:

```
$ sautrela DHMM -En myErgodicHMM -s X,Y 3
<?xml version="1.0" encoding="UTF-8" ?>
<WFSA name="myErgodicHMM" size="3" cdbkSize="2" className="edu.gtts.
    sautrela.wfsa.models.DHMM">
  <IniState name="0" linProb="0.3346555739800134"/>
  <IniState name="1" linProb="0.3334020714858261"/>
  <IniState name="2" linProb="0.3319423545341605"/>
  <FinState name="0" linProb="0.24992777089068277"/>
  <FinState name="1" linProb="0.2497004226995606"/>
  <FinState name="2" linProb="0.2500703421609556"/>
  <Transition from="0" to="0" linProb="0.25025153532497846"/>
  <Transition from="0" to="1" linProb="0.24925182211023583"/>
  <Transition from="0" to="2" linProb="0.25056887167410274"/>
  <Transition from="1" to="0" linProb="0.24983285492067306"/>
  <Transition from="1" to="1" linProb="0.2494480619100129"/>
  <Transition from="1" to="2" linProb="0.2510186604697541"/>
  <Transition from="2" to="0" linProb="0.24934523881987652"/>
  <Transition from="2" to="1" linProb="0.24978910692026185"/>
  <Transition from="2" to="2" linProb="0.25079531209890604"/>
  <Emission state="0" symbol="X" linProb="0.5"/>
  <Emission state="0" symbol="Y" linProb="0.5"/>
  <Emission state="1" symbol="X" linProb="0.5"/>
  <Emission state="1" symbol="Y" linProb="0.5"/>
  <Emission state="2" symbol="X" linProb="0.5"/>
  <Emission state="2" symbol="Y" linProb="0.5"/>
</WFSA>
```

Comando `GMM`

Crea o edita un modelo de mezcla de gaussianas (GMM, Gaussian Mixture Model). Los GMMs son componentes de modelado ampliamente utilizados en las tecnologías del habla, debido a su capacidad de aproximar cualquier densidad de probabilidad. Un GMM puede estar contenido en un WFSA, como en el caso de los HMM continuos, en los cuales cada estado lleva asociado un GMM. Pero también puede ser utilizado como un modelo independiente. En tareas de verificación del locutor y verificación de

la lengua, por ejemplo, es muy común el uso de un modelo universal (UBM, Universal Background Model). Un UBM no es sino un GMM compuesto por multitud de componentes (dependiendo de la aplicación puede rondar las 1024 componentes) entrenado sobre un conjunto extenso de muestras.

Un GMM se compone de una mezcla ponderada de Gaussianas multivariantes, cada una de ellas con sus correspondientes vectores de media y varianza[18], vectores cuya dimensión se corresponde con la dimensión del espacio modelado. Cada Gaussiana establece además un valor mínimo para las mismas (*varFloor*), que es fijado a partir de la primera estimación de las varianzas ($varFloor = var_{ini}/100$). De este modo se evita que las varianzas de las Gaussianas "poco pobladas" tiendan a cero, lo que provocaría problemas numéricos. En la fase de entrenamiento, las Gaussianas poco pobladas serán descartadas, generando nuevas componentes en torno a la más pobladas. Sautrela incorpora el Procesador `GMMTrainer`, que se ocupa de realizar el entrenamiento de un GMM a partir de un conjunto de muestras.

A continuación se muestra la página de manual del comando:

```
$ sautrela -help  GMM

  GMM (edu.gtts.sautrela.wfsa.models.GMM)

    create/resize/modify a Gaussian mixture model (GMM)

  Syntax:  GMM  [-sSmMvVwW] [-n name] [-g gNum] [-d gDim] [URL]

    -s  Turn off high weight Gaussian splitting and discard orphans.
    -S  Turn on high weight Gaussian splitting and discard orphans (by
    default).
    -m  Turn off training of means.
    -M  Turn on training of means (by default).
    -v  Turn off training of variances.
    -V  Turn on training of variances (by default).
    -w  Turn off training of weights.
    -W  Turn on training of weights (by default).
    -n name  Name for the GMM (default: "NoName")
    -g gNum  Number of Gaussians (default: 1)
    -d gDim  Dimension of Gaussians (default: 1)
    URL  Locator of an existing GMM
```

El comando `GMM` permite crear un GMM nuevo o modificar la configuración de un GMM ya existente: la dimensión de las Gaussianas, el número de componentes y los parámetros de entrenamiento. Cuando se crea un GMM nuevo, las medias y varianzas son inicializadas a 0 y 1 respectivamente, mientras que todos los parámetros de entrenamiento quedan activados por defecto. Si se modifica la dimensión de las componentes de un GMM existente, los vectores de medias y varianzas son redimensionados (en caso de aumentar, las nuevas medias y varianzas toman los valores 0 y 1 respectivamente, mientras que en caso de disminuir, los vectores son truncados). La modificación del

[18] La implementación `GMM` de Sautrela hace uso de matrices de covarianza diagonal, con la intención de mejorar la robustez del modelo y evitar problemas de sobreentrenamiento.

número de componentes (el tamaño del GMM) se realiza en función del peso de cada componente. En caso de reducir el tamaño, se eliminan las componentes de menor peso y se renormalizan los pesos de las componentes supervivientes. En caso de aumentar el tamaño, se toma sucesivamente la componente de mayor peso y se le aplica el proceso denominado *splitting*, consistente en generar una nueva componente mediante la perturbación de su vector de medias y asignar a las dos componentes resultantes la mitad del peso de la componente original. Este proceso es repetido hasta alcanzar el tamaño deseado.

Comando `CHMM`

Crea un HMM continuo con topología ergódica o lineal con bucles. Cada estado lleva asociado un GMM, el cual puede ser creado a partir de un GMM existente (a cada estado se le asigna una copia del GMM cargado) o inicializado con una única componente de media cero y varianza unidad. Las probabilidades correspondientes a la topología (probabilidades iniciales, finales y de transición) son inicializadas de acuerdo a una distribución uniforme, salvo en el caso de la topología ergódica, en la cual se añade una pequeña perturbación a dichas probabilidades para romper la simetría del modelo[19]. A continuación se muestra la página de manual del comando:

```
$ sautrela -help CHMM

  CHMM (edu.gtts.sautrela.wfsa.models.CHMM)

    creates a continuous Hidden Markov Model using an existing
    GaussianMixture or an initial zero-mean unity-variance single Gaussian
     per state.

  Syntax:  CHMM  [-E] [-n name] [ -d gDim | -g URL ] size

    -E  Use an ergodic topology. Default topology is linear with loops.
    -n name  The name of the CHMM
    -d gDim  Feature space dimension (default: 1).
    -g URL  Locator of an existing GaussianMixture.
    size  Number of states
```

Comando `CHMMEdit`

Permite modificar todos los parámetros de un HMM continuo. Más en concreto, permite cambiar el nombre del modelo, activar o desactivar el entrenamiento de las transiciones y modificar las propiedades de las componentes Gaussianas de los estados (análogamente a lo mostrado en el comando `GMM`). A continuación se muestra la página de manual del comando:

[19] Como ya se ha mencionado en el comando `HMM`, un modelo ergódico simétrico es equivalente a otro modelo con un único estado. La simetría conlleva la no especialización de los estados (las re-estimaciones derivadas del entrenamiento son idénticas para cada uno de ellos).

```
$ sautrela -help CHMMEdit

  CHMMEdit (edu.gtts.sautrela.wfsa.models.CHMMEdit)

    resize/edit a continuous Hidden Markov Model (CHMM)

  Syntax:  CHMMEdit  [-sSmMtTvVwW] [-n name] [-g gNum] URL

    -s  Turn off high weight Gaussian splitting and discard orphans.
    -S  Turn on high weight Gaussian splitting and discard orphans.
    -m  Turn off training of means.
    -M  Turn on training of means.
    -t  Turn off training of transitions.
    -T  Turn on training of transitions.
    -v  Turn off training of variances.
    -V  Turn on training of variances.
    -w  Turn off training of weights.
    -W  Turn on training of weights.
    -n name  New name for the CHMM
    -g gNum  Number of Gaussians
    URL  Locator of an existing CHMM
```

Comando `CHMMSetEdit`

Modifica los parámetros de todos los HMMs continuos contenidos en un `WFSASet`. En caso de aportar una lista de nombres para los nuevos modelos, se genera un conjunto nuevo de HMMs a partir del conjunto existente. A tal efecto, se van tomando uno a uno en el mismo orden indicado en el descriptor XML y se van añadiendo copias de los modelos existentes pero con los nuevos nombres. Si el tamaño de la lista fuera mayor que el del conjunto de HMMs, se continúa cíclicamente a partir del primero de los modelos.

A continuación se muestra la página de manual del comando:

```
$ sautrela -help CHMMSetEdit

  CHMMSetEdit (edu.gtts.sautrela.wfsa.models.CHMMSetEdit)

    resize/edit a WFSASet of Continouous Hidden Markov Models (CHMM)

  Syntax:  CHMMSetEdit  [-sSmMtTvVwW] [-n nameList] [-g gNum] URL

    -s  Turn off high weigth Gaussian spliting and discard orphans.
    -S  Turn on high weigth Gaussian spliting and discard orphans.
    -m  Turn off means training.
    -m  Turn off training of means.
    -M  Turn on training of means.
    -t  Turn off training of transitions.
    -T  Turn on training of transitions.
```

```
    -v  Turn off training of variances.
    -V  Turn on training of variances.
    -w  Turn off training of weights.
    -W  Turn on training of weights.
    -n nameList  Comma separated list of names for new CHMMs
    -g gNum  Number of Gaussians
    URL  Locator of an existing WFSASet of CHMMs
```

El mecanismo de generación de un conjunto de HMM continuos mediante una lista de nombres resulta útil en varias situaciones. Si partimos de un conjunto que contenga un HMM prototipo, podemos crear un nuevo conjunto a partir de dicho prototipo. En el siguiente caso, por ejemplo, se parte de un `CHMM` prototipo (`proto.chmm`) con el cual se crea un conjunto que se utiliza para generar el conjunto final que contendrá un modelo para cada una de las vocales y un modelo adicional para el silencio.

```
$ sautrela WFSASet proto.chmm > mini.chmmset
$ sautrela CHMMSetEdit -n a,e,i,o,u,sil mini.chmmset > vowels.chmmset
```

Como un `WFSASet` no puede contener dos modelos con un mismo nombre, si la lista contiene un nombre repetido, el correspondiente modelo no será añadido al conjunto resultante. Esta característica puede ser utilizada para hacer una expansión discriminatoria de un conjunto previo. Supongamos que los modelos del ejemplo anterior ya han sido entrenados, pero deseamos generar nuevos modelos para diferentes contextos de aparición de las vocales, pero no así para el silencio. El siguiente ejemplo mantendría todos los modelos originales y generaría dos nuevas variantes solo para los modelos de las vocales:

```
$ sautrela CHMMSetEdit \
    -n a,e,i,o,u,sil,a1,e1,i1,o1,u1,sil,a2,e2,i2,o2,u2 \
    vowels.chmmset > vowels2.chmmset
```

Comando `KTLSS`

Crea o modifica un modelo K-testable en sentido estricto [174, 26, 179]. Si no se indica un localizador de un KTLSS existente, crea un KTLSS vacío que podrá entrenarse posteriormente. A continuación se muestra la página de manual del comando:

```
$ sautrela -help KTLSS

  KTLSS (edu.gtts.sautrela.wfsa.models.KTLSS)

    create/modify a KTLSS language model

  Syntax:  KTLSS  [-gG] [-n name] [-N nGram] [-p pruneCount] [-v vocSize]
    [URL]

    -g  Turn off the growable property
```

```
    -G  Turn on the growable property (default). The KTLSS topology will
    grow up during training
    -n name  The name of the KTLSS
    -N nGram  N value of ngrams
    -p pruneCount  Minimum number of counts for not pruning (1 or lower
    for no pruning)
    -v vocSize  Estimated size of vocabulary
    URL  Locator of an existing KTLSS
```

Comando `TreeModel`

Genera un TreeModel a partir de un WFSA determinista que representa un modelo de lenguaje y un diccionario que puede contener transcripciones múltiples. El autómata resultante es un WFSA no-determinista que integra un modelo de lenguaje y un léxico en árbol, y permite una decodificación más rápida. La Sección 5.7 contiene una descripción más detallada de este modelo. A continuación se muestra la página de manual del comando:

```
$ sautrela -help TreeModel

  TreeModel (edu.gtts.sautrela.wfsa.models.TreeModel)

    creates a TreeModel

  Syntax:  TreeModel  [-d LANGUAGE|LEXICON] name langURL dictURL

    -d LANGUAGE|LEXICON  decoding level (default: LANGUAGE)
    name  The name of the TreeModel
    langURL  Locator of an existing DWFSA to use as language model
    dictURL  Locator of an existing Dictionary
```

Comando `LMM`

Crea un modelo de Markov por capas (*Layered Markov Model*, LMM) [121] a partir de conjuntos de WFSAs. Cada capa se compondrá de un conjunto de WFSAs, salvo la capa superior, que contendrá un único WFSA. Es posible, además, aplicar una transformación afín (un escalado y un desplazamiento) a los logaritmos de las probabilidades de cada capa, lo que puede interpretarse como una ponderación de modelos y una penalización de las inserciones, respectivamente[20]. Un LMM permite formalizar mediante un único WFSA diferentes casos de uso de los WFSAs, lo que permite el uso de un único módulo de decodificación (Procesador `Decoder`) y otro de entrenamiento (Procesador `Trainer`) que dan cobertura a todos los casos. La Sección 5.4 contiene una descripción más extensa de los LMMs.

A continuación se muestra la página de manual del comando:

[20] La penalización (o bonificación, si el desplazamiento fuera positivo) se define capa a capa.

```
$ sautrela -help LMM

  LMM (edu.gtts.sautrela.wfsa.models.LMM)

    create a Layered Markov Model from WFSASets

  Syntax:  LMM  [-c]  [-d dLayer]  [-n name]  <-e | -s | wfsaURL [a b]>
               <wfsasetURL [a b]> ...

    -c  Check the consistency of the connected layers
    -d dLayer  The index [0,n-1] (bottom-up)) of the Layer to be used for
    decoding (default to n-1, top-layer)
    -n name  The name of the LMM
    -e  Equiprobable top Layer (accepts any combination of lower layer
    models)
    -s  Selector top Layer (accepts any single model from the lower layer)
    wsfa  The locator of a WFSA to be used as top Layer
    wsfaset  The locator of a WFSASet for the next Layer
    a  Log scale applied to this layer probabilities
    b  Log offset applied to this layer probabilities
```

4.7.6. Procesadores

Procesador `Decoder`

Decodifica las secuencias de observaciones de entrada utilizando un único WFSA. El resultado de la decodificación es la secuencia de nombres de la secuencia de transiciones (camino) más probable. El procesador desconoce la naturaleza interna del modelo usado para la decodificación así como de las transiciones obtenidas. Se limita a obtener la secuencia de transiciones más probable (haciendo uso únicamente de la interfaz que implementan los modelos) y a consultar el nombre de dichas transiciones [122, 123]. La decodificación puede hacer uso de un ancho de haz (*beam*), descartando durante la búsqueda aquellos estados cuya probabilidad diste demasiado del estado más probable. Las Secciones 5.1 y 5.4 contienen una descripción más extensa del proceso de decodificación y los LMMs, respectivamente.

Procesador `Trainer`

Entrena un WFSA o un conjunto de capas de conocimiento representadas por conjuntos de WFSAs a partir de las observaciones de entrada. El entrenamiento de los WFSAs se lleva a cabo de una manera similar a la decodificación, ya que el módulo de entrenamiento desconoce la naturaleza interna de los modelos [124, 126]. Siguiendo el esquema del algoritmo esperanza-maximización (EM), el módulo entrenador estima la esperanza de todas las posibles transiciones a las que da lugar la secuencia de observaciones de entrada (haciendo uso de la interfaz de decodificación), y dicha esperanza es transmitida al modelo, quien se ocupa de realizar las estimaciones de máxima verosimilitud de los parámetros internos. De manera análoga al procesador de deco-

dificación, es posible hacer uso de un ancho de haz que limite el número de estados activos en cada instante. Además, la esperanza de las transiciones puede ser obtenida a partir del camino (secuencia de transiciones) más probable o de todos los posibles caminos, dando lugar a los entrenamientos comúnmente denominados como Viterbi y Baum-Welch, respectivamente.

Para el caso del entrenamiento de capas de conocimiento, toma de la cabecera de cada secuencia de datos una transcripción con la cual construir una capa adicional de supervisión. El procesador permite configurar la propiedad que contiene la transcripción y permite cierto grado de libertad en cuanto a la forma de obtener la capa de supervisión a partir de dicha transcripción.

Las Secciones 5.2 y 5.6 contienen una descripción más extensa del proceso de entrenamiento.

Procesador `GMMTrainer`

Entrena un GMM a partir de las muestras de entrada. Este procesador permite además acelerar el entrenamiento de un GMM mediante dos vías complementarias. Por un lado, dispone de la posibilidad de crear hilos de ejecución que realizarán el entrenamiento en paralelo. Cada secuencia de datos es transmitida a uno de los hilos de ejecución, que contiene una copia del GMM (evitando así la gestión del acceso a recursos compartidos) y, una vez finalizado el procesamiento de los datos de entrada, los parámetros de entrenamiento son recombinados en un único GMM. Por otro lado, permite una estimación rápida (pero aproximada) de las N componentes más probables dada una observación de entrada, limitando en cada instante el entrenamiento a dichas componentes. Este método puede resultar de gran utilidad cuando se entrenan GMMs de gran tamaño, como ocurre en las tareas de verificación del locutor y de la lengua, donde suelen entrenarse GMMs de más de 500 componentes.

Procesador `RandomSymbolGenerator`

Toma un WFSA y genera secuencias aleatorias de símbolos. Independientemente de la naturaleza de los símbolos, que de penderá del modelo utilizado, cada símbolo es convertido a su representación textual, por lo que volcará a su salida una secuencia de datos textuales.

Procesador `CrossEntropyEstimator`

Toma un WFSA y realiza una estimación empírica de la entropía cruzada [196] para el conjunto de secuencias de entrada. Esta magnitud indica la cantidad promedio de bits necesarios para predecir o codificar una observación de entrada dado un modelo probabilístico, y está directamente relacionada con la verosimilitud y la Perplejidad [205].

La entropía cruzada empírica puede utilizarse con dos fines. Por un lado, sirve para, dado un conjunto de secuencias de entrada, comparar diferentes modelos probabilísticos (o variaciones en su estimación), ya que nos da una idea de cómo de bien predice un cierto modelo las secuencias de entrada. Por otro lado, también es posible establecer un modelo probabilístico como referencia y modificar la parametrización de las secuencias

de entrada, con lo cual mediríamos la capacidad que tiene cada parametrización de filtrar el ruido (la variabilidad no deseada) de los datos de entrada[21].

Procesador `RecognitionRate`

Compara la salida de un decodificador frente a la transcripción contenida en las cabeceras de los datos y estima una tasa de reconocimiento. Ofrece la posibilidad de aplicar distintas funciones de evaluación (tasas de reconocimiento), cada una de ellas definida mediante su correspondiente función de edición (también denominada distancia de edición en aquellos casos en los que represente un *grado de error*, y se trate, por tanto, de una función a minimizar[22]). En aquellos casos en los que la función de evaluación esté basada en una distancia de edición no normalizada (la función de coste no contiene una normalización por la longitud del camino), el problema de optimización (el alineamiento de la secuencia de referencia y la secuencia reconocida) es resuelto mediante programación dinámica. En los casos en los que la función de evaluación esté basada en una distancia de edición normalizada (la función de coste contiene una normalización por la longitud del camino), el problema de optimización es resuelto mediante la programación fraccional [109, 187, 144].

4.7.7. Ejemplo de uso: Entrenamiento de modelos acústicos

Para el siguiente ejemplo tomaremos como referencia la base de datos acústica `MyAcousticDataBase.xml` del Listado 4.3. El primer paso será la parametrización de la base de datos, para lo que usaremos la Engine `ASRParam16k.eng` del Listado 4.4, que permite volcar a un fichero un subconjunto parametrizado de la base de datos. Se volcarán dos subconjuntos, el subconjunto de aprendizaje (aquellos audios cuyos nombres comienzan por `wav_ac_train`), que será utilizado para entrenar los modelos, y el subconjunto de evaluación (aquellos audios cuyos nombres comienzan por `wav_ac_eval`), que será utilizado para obtener el índice de precisión de los modelos:

```
$ sautrela ASRParam16k.eng -d MyAcousticDataBase.xml \
    -s "wav_ac_train.*" -o train.dump
$ sautrela ASRParam16k.eng -d MyAcousticDataBase.xml \
    -s "wav_ac_eval.*" -o eval.dump
```

Los modelos acústicos utilizados serán HMM continuos (CHMM). Una de las maneras de crearlos es estimando inicialmente una única componente Gaussiana con todas las muestras de entrenamiento. Para ello generamos primero un GMM con una única componente de dimensión 39 (la parametrización utilizada genera 13 cepstrales, más

[21] Nótese que una vez se ha establecido un modelo probabilístico de referencia, toda variabilidad que sí contenga información relevante pero que no sea capturable por el modelo es implícitamente considerada como variabilidad ruidosa o no deseada. Es decir, fijado un modelo de referencia, podría darse el caso de que una parametrización que sí incluya mucha información relevante sea evaluada como *pobre*, debido a que gran parte de dicha información no puede ser capturada por el modelo.

[22] Para toda función de edición que represente un *grado de acierto* (una función a maximizar), existe su versión *grado de error* equivalente, por lo que todos los problemas basados en funciones de edición pueden ser reducidos a optimizaciones de problemas basados en distancias de edición.

Listado 4.16 `GMMTrainer.eng` - Descriptor XML de una engine de entrenamiento de GMMs. Los datos de entrada son cargados desde un fichero.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE Engine SYSTEM "http://sautrela.org/lib/Engine.dtd">
<Engine name="GMMTrainer" description="Trains a GMM from a dumped data file">
  <Processor name="StreamReader">
    <param name="streamURL" value="?-d [data.dump]" />
  </Processor><Buffer/>
  <Processor name="GMMTrainer">
    <param name="modelURL" value="?-i [in.gmm]" />
    <param name="outputFile" value="?-o [out.gmm]" />
    <param name="threadNumber" value="?-t" />
    <param name="topN" value="?-n" />
  </Processor>
</Engine>
```

sus primeras y sus segundas derivadas) que entrenamos a continuación mediante la Engine `GMMTrainer.eng` del Listado 4.16:

```
$ sautrela GMM -g 1 -d 39 > gmm
$ sautrela GMMTrainer.eng -d ../param/train.dump -i gmm -o gmm
```

Una vez estimada la Gaussiana, creamos un conjunto de CHMMs compuesto únicamente por el modelo prototipo:

```
$ sautrela CHMM -g gmm 3 > proto.chmm
$ sautrela WFSASet proto.chmm > proto.chmmset
```

A continuación, obtenemos el inventario de fonemas para poder generar un modelo acústico por fonema:

```
$ sautrela AcousticDataBase -nup Phon MyAcousticDataBase.xml ".*" \
    | cut -d " " -f 3- | tr " " "\n" | sort | uniq > phones.txt
```

Ya estamos en condiciones de crear el conjunto de modelos acústicos, que estará formado por el inventario fonético extraído de las transcripciones y el fonema de silencio, que en nuestro caso no aparecía en las transcripciones:

```
$ sautrela CHMMSetEdit -n "$(paste -s -d, phones.txt),SIL" \
    proto.chmmset > phones.1g.chmmset
```

Una vez creado el conjunto de modelos, podemos iniciar el entrenamiento de los mismos. Para ello utilizaremos la Engine `WFSATrainer.eng` del Listado 4.17 que entrenará el conjunto de modelos haciendo uso de la transcripción disponible en el subconjunto de entrenamiento de la base de datos. El siguiente comando ejecuta una iteración de entrenamiento del conjunto de modelos acústicos:

```
$ sautrela WFSASetTrainer.eng -d train.dump \
    -i phones.1g.chmmset -o phones.1g.chmmset
```

Listado 4.17 `WFSATrainer.eng` - Descriptor XML de una engine de entrenamiento de WFSAs. Durante el entrenamiento se permite la inserción opcional del símbolo `SIL` en la transcripción obtenida a partir de la propiedad `Phon`.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE Engine SYSTEM "http://sautrela.org/lib/Engine.dtd">
<Engine name="WFSA/WFSASets Trainer" description="Trains WFSA/WFSASets from a
    dumped data file.">
  <Processor name="StreamReader">
    <param name="streamURL" value="?-d" />
  </Processor><Buffer/>
    <Processor name="Trainer">
    <param name="modelURLList" value="?-i" />
    <param name="outputFileList" value="?-o" />
    <param name="beam" value="?-beam" />
    <param name="transcInsertionMask" value="?-tim [lr]" />
    <param name="transcInsertionSymbol" value="?-tis [SIL]" />
    <param name="transcPropertyName" value="?-tpn [Phon]" />
    <param name="verbose" value="?-v" />
  </Processor>
</Engine>
```

Nótese que la Engine `WFSATrainer.eng` modifica algunos de los valores por defecto, adaptándose al ejemplo que nos ocupa. El nombre por defecto de la propiedad que contiene la transcripción (parámetro `transcPropertyName`) es fijado a `Phon`. Se modifica el valor por defecto del parámetro `transcInsertionMask`, que establece de qué manera puede ser modificada la transcripción para incluir inserciones del símbolo indicado por el parámetro `transcInsertionSymbol`. La inserción de un símbolo especial permite considerar ciertos eventos no incluidos en la transcripción (típicamente, silencios). La máscara de inserción puede contener los caracteres `'l'` (*left* o inserción inicial), `'r'` (*right* o inserción final) o `'i'` (*intermediate* o inserción entre los elementos de la transcripción). En concreto, la configuración por defecto de la engine permite la inserción inicial y final del modelo `SIL`, algo más que razonable, ya que equivale a suponer que las señales de audio pueden contener un segmento inicial y final de silencio.

El comando anterior debe ser ejecutado hasta superar algún criterio de convergencia, pero en el presente ejercicio supondremos que diez iteraciones de entrenamiento son suficientes para cualquier fase de entrenamiento. Para simplificar la sintaxis de los comandos, crearemos una sencilla función `repeat` que ejecuta un comando tantas veces como se indique:

```
$ function repeat { n=$1; shift; for ((i=0;i<n;i++)); do "$@" ; done; }
```

Utilizaremos la función `repeat` para reescribir el comando que, ahora sí, realiza 10 iteraciones de entrenamiento:

```
$ repeat 10 sautrela WFSATrainer.eng -d train.dump \
    -i phones.1g.chmmset -o phones.1g.chmmset
```

Una vez superada la primera fase de entrenamiento, contamos con HMMs continuos

Listado 4.18 `WFSARates.eng` - Descriptor XML de una engine de obtención de tasas a partir de un conjunto de WFSAs. La transcripción de referencia para la obtención de las tasas es extraída de la propiedad `Phon`, y el símbolo `SIL` es ignorado.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE Engine SYSTEM "http://sautrela.org/lib/Engine.dtd">
<Engine name="WFSARates"
   description="Estimates the accuracy for a WFSA/WFSASet.">
   <Processor name="StreamReader">
      <param name="streamURL" value="?-d [data.dump]" />
   </Processor><Buffer/>
   <Processor name="Decoder">
      <param name="beam" value="?-beam" />
      <param name="modelURL" value="?-i" />
      <param name="verbose" value="?-v" />
   </Processor><Buffer/>
   <Processor name="RecognitionRate">
      <param name="excludeSymbolPattern" value="?-e [SIL]" />
      <param name="transcPropertyName" value="?-p [Phon]" />
   </Processor>
</Engine>
```

de tres estados y una única componente Gaussiana por estado, entrenados a partir de las transcripciones y posibles silencios en los bordes. Si suponemos que el modelo de silencio es ya mínimamente robusto, podemos dar más libertad a la inserción de silencios, permitiendo también la inserción libre de silencios entre los fonemas de la transcripción[23]. Una vez que ya contamos con modelos iniciales entrenados (aunque cada estado contiene una sola componente, esta ya se ha especializado), también es recomendable acelerar el entrenamiento mediante la búsqueda en haz, que descartará en cada instante los estados que resulten muy improbables, para reducir así el espacio de búsqueda. El valor del parámetro `beam` es la diferencia máxima (en escala logarítmica) aceptable entre el estado más probable y el de menor probabilidad. Se trata de un valor heurístico que debe ser ajustado *ad hoc*, en función de la naturaleza de los modelos y la base de datos que estemos utilizando para entrenarlos[24]. Una de las mejores maneras de ajustarlo es mediante prueba y error, monitorizando bien el número promedio de estados supervivientes (valor de verbosidad 2) o bien la secuencia total de estados supervivientes (valor de verbosidad 3). En el presente ejemplo supondremos que un ancho de haz de 50 es adecuado en todos los casos. La siguiente fase de entrenamiento podría basarse en el siguiente comando:

```
$ repeat 10 sautrela WFSATrainer.eng -d train.dump \
    -i phones.1g.chmmset -o phones.1g.chmmset -tim lir -beam 50
```

La Engine `WFSASetRates.eng` del Listado 4.18 permite obtener el valor del índice

[23] A pesar de que pudiera haberse hecho desde el principio, por cuestiones de convergencia, resulta más adecuado hacerlo una vez se cuente con modelos de silencio mínimamente entrenados.

[24] Si el valor del ancho de haz es muy pequeño, podría darse el caso de que dada una secuencia de observaciones de entrada no exista ningún estado final activo para la última observación. En tal caso, el entrenador vuelve a iniciar el proceso de entrenamiento de esa secuencia de datos, duplicando el ancho de haz.

de precisión fonético (*phone accuracy*) de los modelos entrenados:

```
$ sautrela WFSASetRates.eng -d train.dump -i phones.1g.chmmset -beam 50
```

Ya solo queda aumentar la complejidad de los modelos, duplicando el número de componentes y reentrenándolos hasta obtener el tamaño deseado[25]:

```
$ sautrela CHMMSetEdit -g 2 phones.1g.chmmset > phones.2g.chmmset
$ repeat 10 sautrela Trainer.eng -d train.dump \
    -i phones.2g.chmmset -o phones.2g.chmmset -m lir -beam 50
$ sautrela WFSASetRates.eng -d train.dump -i phones.2g.chmmset -beam 50
$ sautrela CHMMSetEdit -g 4 phones.2g.chmmset > phones.4g.chmmset
$ repeat 10 sautrela Trainer.eng -d train.dump \
    -i phones.4g.chmmset -o phones.4g.chmmset -m lir -beam 50
$ sautrela WFSASetRates.eng -d train.dump -i phones.1g.chmmset -beam 50
$ ...
```

4.7.8. Ejemplo se uso: Generación aleatoria de poemas

En el presente ejemplo, tomaremos el poema *"Llegó con tres heridas"* de Miguel Hernández (Listado 4.19) como corpus de entrenamiento de un modelo de lenguaje KTLSS, con el cual se generarán nuevos poemas aleatorios. Comprobaremos que en función de la complejidad del modelo entrenado, el resultado será más cercano a un poema real. El comando `KTLSS` permite crear un modelo de lenguaje inicialmente vacío (sin conocimiento a priori) que podrá ser reentrenado. Cuando creamos un modelo KTLSS inicial debemos asignar dos valores relevantes: la profundidad de n-grama (las probabilidades del modelo se basarán en secuencias históricas de tamaño $n-1$) y el tamaño estimado de vocabulario (un modelo KTLSS da cobertura a palabras fuera de vocabulario, y para ello precisa un valor estimado del tamaño del vocabulario). En el presente ejemplo, crearemos modelos de 1-gramas, 2-gramas y 3-gramas, y se utilizará un tamaño de vocabulario de 20:

```
$ sautrela KTLSS -v 20 -N 1 > 1gram.ktlss
$ sautrela KTLSS -v 20 -N 2 > 2gram.ktlss
$ sautrela KTLSS -v 20 -N 3 > 3gram.ktlss
```

Para entrenar los tres modelos de lenguaje usaremos la Engine `WFSATxtTrainer.eng` del Listado 4.20. Esta engine está compuesta por dos procesadores. El primero (`TextReader`) permite cargar un recurso de texto, procesando cada línea del recurso y generando una secuencia de `StringData` por cada línea (cada `StringData` contendrá una palabra[26]). Durante el procesamiento previo permite eliminar subcadenas (aquellas que coincidan con la expresión regular aportada), lo

[25] También es posible generar primero el tamaño deseado y después entrenar el conjunto de WFSAs pero, por cuestiones de convergencia, resulta más adecuado hacerlo de una manera escalonada.

[26] El Procesador `TextReader` tiene un parámetro adicional (`splitRegexp`) que permite configurar la expresión regular utilizada para trocear las líneas de texto (su valor por defecto es `"\s"`, cualquier carácter de espaciado: espacio, tabulador, salto de carro, etc.)

Listado 4.19 Fichero de texto `MHernandez.txt` que contiene el poema *"Llegó con tres heridas"* de Miguel Hernández.

```
Llegó con tres heridas:
la del amor,
la de la muerte,
la de la vida.
Con tres heridas viene:
la de la vida,
la del amor,
la de la muerte.
Con tres heridas yo:
la de la vida,
la de la muerte,
la del amor.
```

Listado 4.20 `WFSATxtTrainer.eng` - Descriptor XML de una engine de entrenamiento de un WFSA a partir de un recurso de texto. La engine ofrecerá además la posibilidad de eliminar aquellas secciones del texto original que coincidan con una expresión regular, así como la posibilidad de convertirlo a mayúsculas o minúsculas.

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE Engine SYSTEM "http://sautrela.org/lib/Engine.dtd">
<Engine name="txt2WFSA" description="trains a WFSA from text input">
   <Processor name="TextReader">
      <param name="charset" value="?-c"/>
      <param name="textURL" value="?-txt"/>
      <param name="deleteRegex" value="?-d"/>
      <param name="splitRegex" value="?-s"/>
      <param name="toUpperCase" value="?-u"/>
      <param name="toLowerCase" value="?-l"/>
   </Processor><Buffer/>
   <Processor name="Trainer">
      <param name="modelURLList" value="?-i"/>
      <param name="outputFileList" value="?-o"/>
   </Processor>
</Engine>
```

Listado 4.21 `RandomSymbols.eng` - Descriptor XML de una engine de generación aleatoria de cadenas de símbolos. El resultado es mostrado por la salida estándar.

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE Engine SYSTEM "http://sautrela.org/lib/Engine.dtd">
<Engine name="RandomSymbols" description="create random symbol sentences from a
    WFSA">
  <Processor name="RandomSymbolGenerator">
    <param name="modelURL" value="?-i"/>
    <param name="streamNumber" value="?-n"/>
  </Processor><Buffer/>
  <Processor name="DataJoiner"/><Buffer/>
  <Processor name="Sniffer">
    <param name="mode" value="TEXT"/>
  </Processor>
</Engine>
```

que en nuestro caso nos servirá para eliminar los símbolos de puntuación (`"[\.,:]"`). También permite pasar todo el texto a mayúsculas o a minúsculas. El segundo procesador es el entrenador (`Trainer`) utilizado en ejemplos anteriores y que en este caso se utilizará para entrenar un único WFSA (el modelo de lenguaje correspondiente). Mediante las siguientes líneas de comando realizará el entrenamiento de los tres modelos:

```
$ sautrela WFSATxtTrainer.eng -txt MHernandez.txt -d "[\.,:]" -l true \
    -i 1gram.ktlss -o 1gram.ktlss
$ sautrela WFSATxtTrainer.eng -txt MHernandez.txt -d "[\.,:]" -l true \
    -i 2gram.ktlss -o 2gram.ktlss
$ sautrela WFSATxtTrainer.eng -txt MHernandez.txt -d "[\.,:]" -l true \
    -i 3gram.ktlss -o 3gram.ktlss
```

Una vez entrenados los modelos, haremos uso de la Engine `RandomSymbols.eng` del Listado 4.21. Esta engine comienza con el Procesador `RandomSymbolGenerator` que toma un WFSA y lo utiliza para generar secuencias aleatorias. Cada secuencia estará compuesta por un número indeterminado de elementos `StringData` que serán concatenados en una única cadena de caracteres mediante el Procesador `DataJoiner` y, finalmente, serán mostradas por la salida estándar mediante el Procesador `Sniffer`. A continuación se muestra la salida del comando que creará un poema aleatorio a partir del modelo de 1-gramas:

```
$ sautrela RandomSymbols.eng -i 1gram.ktlss -n 12
heridas la con
amor heridas la de yo
muerte llegó de
del
con de la con vida la la la yo amor heridas del la tres de tres de la
la la amor
la la de muerte la con la con con
tres
vida la amor la
```

```
de la
muerte de amor
heridas vida heridas muerte la de la de del la la amor heridas la
```

Para un modelo de 1-gramas, la probabilidad de un símbolo es independiente de la historia previa (la longitud de la historia considerada es de 0), por lo que no puede apreciarse ninguna construcción gramatical, más allá del mero azar. Sin embargo, al utilizar el modelo de 2-gramas, las secuencias comienzan a tener cierto grado de estructura secuencial, ya que cada nuevo símbolo aleatorio dependerá de cuál fue el anterior (la longitud de la historia modelada es 1). Ello, no obstante, no puede evitar (en cierta manera promueve) la generación de estructuras secuenciales repetitivas como las del ejemplo (`"la de la de la de la"`):

```
$ sautrela RandomSymbols.eng -i 2gram.ktlss -n 12
heridas yo
amor
la vida con tres heridas
de la muerte
con tres heridas
vida
la muerte con tres heridas yo
tres heridas
de la del amor
la de la de la de la vida
con tres heridas
tres heridas heridas viene
```

El modelo de 3-gramas, como es de esperar, es capaz de generar estructuras secuenciales más complejas, en las que la probabilidad de un nuevo símbolo dependerá de los dos símbolos anteriores. El resultado podría llegar a tener incluso cierta estética artística:

```
$ sautrela RandomSymbols.eng -w 3gram.ktlss -n 12
heridas yo
amor
la vida
muerte con tres heridas yo
con tres heridas viene
la de la del amor
del amor
de tres heridas yo
la de la muerte
la de la vida
con tres heridas
tres heridas
```

De hecho, el uso de 3-gramas está muy extendido en las implementaciones de reconocedores del habla continua, debido a su interesante equilibrio entre complejidad y capacidad de modelado.

Capítulo 5

Sautrela: Aportaciones relevantes

El presente capítulo desarrolla de manera más extensa aquellos elementos de Sautrela [122, 123] que resultan de especial interés debido a que contienen aportaciones teóricas o algorítmicas relevantes relacionadas con las Tecnologías del Habla.

La Secciones 5.1 y 5.2 describen los mecanismos unificados de decodificación y re-estimación: todo modelo basado en autómatas de estados finitos ponderados y que implemente la interfaz WFSA, independientemente de su implementación o naturaleza interna, podrá ser integrado en Sautrela bien en un proceso de decodificación, bien en un proceso de re-estimación. La Sección 5.3 profundiza en el proceso de re-estimación de modelos y propone un mecanismo unificado de re-estimación Maximum A Posteriori (MAP).

La Sección 5.4 describe los denominados modelos de Markov por capas (LMM), que integran en un único WFSA distintos niveles o capas de conocimiento representados por conjuntos de WFSAs. A continuación, las Secciones 5.5 y 5.6 analizan el papel que juegan los LMMs en tres aspectos cruciales de la arquitectura de Sautrela: la decodificación acoplada, la re-estimación supervisada y la re-estimación acoplada.

Para finalizar, la Seccion 5.7 describe el modelo TreeModel que integra en un único WFSA un modelo de lenguaje y un conjunto de modelos léxicos en árbol, y cuya decodificación resulta equivalente a la búsqueda en árbol léxico implementada en gran parte de sistemas de RAH.

5.1. Decodificación unificada

Sautrela define un conjunto de interfaces que permite, dado un autómata de estados finitos ponderado, formalizar un mecanismo unificado de decodificación que obtiene la secuencia más probable de transiciones dada una secuencia de observaciones. Por *unificado* nos referimos al hecho de que, gracias a la abstracción funcional que ofrece el conjunto de interfaces diseñadas, es posible definir un único mecanismo de decodificación capaz de trabajar frente a una gran variedad de modelos probabilísticos utilizados

en las Tecnologías del Habla. El Listado 5.1 muestra el conjunto de interfaces[1] sobre el cual se asienta el mecanismo. El contenido de las interfaces puede resumirse en los siguientes puntos:

- Todo autómata (interfaz `WFSA`, *Weighted Finite State Automaton*), símbolo[2] (interfaz `Symbol`) o transición (interfaz `Transition`) cuenta con un método `getName()` que retorna su identidad o nombre.

- Un alfabeto (interfaz `Alphabet`) representa un conjunto de símbolos con nombres diferentes. Todo alfabeto cuenta con un método `valueOf(String name)` que permite obtener un símbolo a partir de su nombre[3].

- Todo autómata lleva asociado un alfabeto, que es retornado por el método `getAlphabet()`.

- Todo autómata tiene un único estado inicial, que es retornado por el método `getIniState()`.

- Todo estado `s` lleva asociada una probabilidad[4] de ser final, que es retornada por el método `getFinProb(State s)` del autómata.

- Una transición está compuesta por un estado origen, un estado destino, un símbolo de emisión y una probabilidad de transición. Toda transición cuenta con los métodos `getSource()`, `getDestination()`, `getSymbol()` y `getProbability()` que retornan cada uno de los cuatro elementos de que se compone, respectivamente.

- Todo autómata cuenta con un método `getTrans(State s)` que retorna el conjunto completo de transiciones[5] que parten de un estado estado origen `s`.

- Todo autómata cuenta con un método `getTrans(State s, Symbol y)` que, dado un estado origen `s` y un símbolo de emisión `y`, retorna una única transición[6] si se trata de un autómata determinista (interfaz `DWFSA`, *Deterministic Weighted Finite State Automaton*), o un conjunto de transiciones[7] si se trata de un autómata no-determinista (interfaz `NdWFSA`, *Non-deterministic Weighted Finite State Automaton*).

[1] En la presente memoria, y para facilitar la lectura, se ha evitado la notación de Genéricos en Java. La documentación completa del código fuente puede ser consultada en `http://www.sautrela.org`.

[2] En la presente memoria, el término símbolo se refiere a una observación del modelo implementado por un WFSA. Puede representar tanto observaciones pertenecientes a un alfabeto finito, como observaciones pertenecientes a un alfabeto infinito (que puede o no representar un espacio continuo). A su vez, puede estar compuesto por un único valor o múltiples valores, así como ser de naturaleza textual o numérica, etc. El uso de interfaces permite a Sautrela interaccionar con todo tipo de símbolo sin necesidad de conocer su naturaleza (implementación).

[3] De no existir símbolo alguno con ese nombre, el método devolverá el valor `null`.

[4] Por cuestiones de eficiencia computacional, la interfaz WFSA determina que las probabilidades vienen dadas en escala logarítmica natural (es más frecuente el cálculo del producto de probabilidades que la suma de las mismas). En la presente memoria, sin embargo, se ignorará este hecho.

[5] De no existir transición alguna, el método devuelve un conjunto vacío.

[6] De no existir transición alguna, el método devuelve el valor `null`.

[7] De no existir transición alguna, el método devuelve un conjunto vacío.

Listado 5.1 Conjunto de interfaces sobre el cual se asienta la decodificación unificada. Para el caso de la interfaz `WFSA`, se muestra el subconjunto de métodos relativos a la decodificación. El presente código fuente evita la notación de Genéricos de Java, pero su documentación completa puede ser consultada en `http://www.sautrela.org`.

```
public interface Named {
   public String getName();
}

public interface State {}

public interface Symbol extends Named {}

public interface Transition extends Named,Comparable {
   public State getSource();
   public State getDestination();
   public Symbol getSymbol();
   public double getProbability();
}

public interface Alphabet extends Iterable {
   public Symbol valueOf(String name);
}

public interface WFSA extends Named {
   ...
   public Alphabet getAlphabet();
   public State getIniState();
   public double getFinProb(State s);
   public Transition[] getTrans(State s);
   ...
}

public interface DWFSA extends WFSA {
   public Transition getTrans(State s,Symbol y);
}

public interface NdWFSA extends WFSA {
   public Transition[] getTrans(State s,Symbol y);
}
```

- Para todo estado, la suma de la probabilidad de ser final y las probabilidades del conjunto de transiciones que parten de él es la unidad[8].

Esta abstracción funcional es suficientemente flexible como para dar cabida a diversos tipos de modelos basados en autómatas y que son habitualmente utilizados en las tecnologías del habla, sin imponer criterios estrictos de implementación. En concreto:

Estados y transiciones: La topología del autómata puede tener una implementación interna estática o dinámica. Los autómatas suelen tender a tener una representación interna estática de todos los estados y las posibles transiciones. Sin embargo en algunos casos los estados pueden ser el resultado de una combinatoria de elementos internos, de tal manera que el número final de estados y transiciones sea intratable. En tales casos, suele optarse por una implementación dinámica que genera los estados o transiciones bajo demanda. La interfaz de decodificación es compatible con ambos tipos de autómatas. Es más, la interfaz definida da cabida incluso a autómatas con un conjunto infinito de estados o transiciones, siempre y cuando el conjunto de transiciones que parte de un estado `s` dado un símbolo `y` sea finito.

Símbolos y alfabeto: Los símbolos pueden ser de naturaleza discreta o continua, tanto escalares como vectoriales. Es posible hacer uso de autómatas que modelan secuencias de caracteres, secuencias de números enteros, secuencias de números reales, secuencias de vectores reales, etc. Partiendo de una representación textual de la secuencia de observaciones, el alfabeto del autómata permite convertir cada observación o muestra de entrada en el correspondiente símbolo interno del autómata. El uso de símbolos de naturaleza continua da lugar a un alfabeto continuo, y por tanto infinito. Ello no es un problema desde el punto de vista de la abstracción funcional definida, ya que en ningún momento se hace uso explícito del conjunto total de símbolos[9]. Los símbolos son una mera referencia a la representación interna que el autómata tiene de las observaciones o muestras de entrada.

A partir de este conjunto de interfaces, la definición de un proceso unificado de decodificación que obtenga la secuencia más probable de transiciones dada una secuencia de observaciones resulta extremadamente sencilla. La abstracción funcional reduce el problema a la definición de un algoritmo de decodificación para cada uno de los tipos de autómatas definidos: el determinista (DWFSA) y el no-determinista (NdWFSA).

El Algoritmo 5.1 muestra los pasos a seguir para la decodificación de una secuencia de símbolos dado un autómata determinista. El algoritmo es extremadamente sencillo,

[8] Entiéndase que, debido al uso de la escala logarítmica, no nos referimos a la suma de los valores retornados por los métodos `getFinProb(State s)` y `getProbability()`, sino a la suma de sus exponenciales.

[9] Aunque la decodificación no hace uso expreso del conjunto total de símbolos, la interfaz `Alphabet` es declarada como iterable (extiende `Iterable`) y, por tanto, debería ser posible recorrer el conjunto completo de símbolos. En el caso de los alfabetos infinitos, Sautrela establece que todo intento de recorrerlos debe generar una excepción de tipo `UnsupportedOperationException`. La decodificación no hace uso de la propiedad iterable de los alfabetos, y su razón de ser está ligada a la composición de modelos (véanse Secciones 5.4 y 5.7).

Algoritmo 5.1 Algoritmo de decodificación de una secuencia de símbolos dado un autómata de estados finitos ponderado determinista (DFWSA). Una transición queda definida por una tupla $t = (s_{src}, s_{dst}, x, p)$ que contiene el estado origen s_{src}, el estado destino s_{dst}, la observación x y la probabilidad de transición p. Las funciones `getInitState` $(\cdot)$ y `getFinProb` $(\cdot)$ corresponden a sendos métodos de la interfaz `WFSA` y el método `getTrans` $(\cdot)$ corresponde a la interfaz `DWFSA`, todos ellos incluidos en el Listado 5.1.

Input:
A, a `DWFSA` (Deterministic Weighted Finite-State Automaton)
$X = (x_1, \ldots, x_N)$, a sequence of `Symbols`
Output:
$T = (t_1, \ldots, t_N)$, a sequence of `Transitions`
Function `decodeDWFSA` (A, X) : T
  $T \leftarrow ()$
  $s \leftarrow$ `getInitState` (A)
  for $x \in X$ **do**
  | $t \leftarrow$ `getTrans` (A, s, x)
  | **if** $t =$ `NIL` **then**
  | | $T \leftarrow ()$
  | | **return**
  | **end**
  | $T \leftarrow T + (t)$
  | $(\sim, s, \sim, \sim) \leftarrow t$
  end
  if `getFinProb` $(A, s) = 0$ **then**
    $T \leftarrow ()$
  end
end

ya que el camino final no es sino el resultado de una consulta secuencial de la transición asociada a cada símbolo. En caso de que no exista un camino de decodificación, el algoritmo devuelve una secuencia vacía (puede darse el caso de que, una vez procesado parte de la secuencia, no exista ninguna transición para la siguiente observación, o que, una vez finalizado el procesamiento de la secuencia, el estado resultante no sea final).

El Algoritmo 5.2 muestra los pasos a seguir para la obtención de la secuencia de símbolos más probable dado un autómata no-determinista. En cada instante (para cada símbolo de entrada) se obtiene el conjunto de posibles caminos a partir del conjunto de caminos del instante anterior. Una vez finalizado el procesamiento de todos los símbolos y aplicadas las probabilidades finales a cada uno de los caminos, se retorna la secuencia de transiciones correspondiente al más probable. El algoritmo incorpora también la denominada búsqueda en haz (*beam search*), descartando tras el procesamiento de cada símbolo de entrada todos aquellos caminos cuyos logaritmos de probabilidad

Algoritmo 5.2 Algoritmo de decodificación de una secuencia de símbolos dado un autómata no-determinista (NdWFSA). Para cada símbolo de entrada x_n, el algoritmo obtiene un mapa M cuyo conjunto de claves representa el conjunto de estados posibles, y los valores son tuplas compuestas por la secuencia de transiciones más probable y el producto de sus probabilidades. La función `prune`$(M, beam)$ elimina del mapa todas las entradas correspondientes a estados cuyo logaritmo de probabilidad diste más de *beam* frente al estado más probable. Las probabilidades se suponen dadas en su forma lineal.

Input:

A, a `NdWFSA` (Non-deterministic Weighted Finite-State Automaton)

$X = (x_1, \ldots, x_N)$, a sequence of `Symbols`

beam, a real number (the search beam as *logProb*)

Output:

$T = (t_1, \ldots, t_N)$, a sequence of `Transitions`

Function `decodeNdWFSA` $(A, Y, beam)$: T

$M_{old} \leftarrow \{\texttt{getInitState}\,(A) \mapsto (()\,, 1)\}$
for $x \in X$ **do**
  $M \leftarrow \{\mapsto\}$
  for $s_{src} \mapsto (T, P) \in M_{old}$ **do**
    for $t \in \texttt{getTrans}\,(A, s_{src},\, x)$ **do**
      $(\sim, s_{dst}, \sim, p) \leftarrow t$
      if $M\,[s_{dst}] = \texttt{NIL}$ **then**
        $M\,[s_{dst}] \leftarrow (T + (t)\ ,\ P \cdot p)$
      else
        $(\sim, p_{best}) \leftarrow M\,[s_{dst}]$
        if $P \cdot p > p_{best}$ **then**
          $M\,[s_{dst}] \leftarrow (T + (t)\ ,\ P \cdot p)$
        end
      end
    end
  end
  $M \leftarrow \texttt{prune}\,(M, beam)$
  if $|M| = 0$ **then**
    $T \leftarrow (\,)$
    return
  end
  $M_{old} \leftarrow M$
end
$(s, (T, \sim)) \leftarrow \underset{s \mapsto (T,p) \in M}{\arg\max}\ \{p + \texttt{getFinProb}\,(A, s)\}$
if $\texttt{getFinProb}\,(A, s) = 0$ **then**
  $T \leftarrow (\,)$
end
end

Algoritmo 5.3 Algoritmo de decodificación de una secuencia de símbolos dado un autómata de estados finitos ponderado no-determinista (NdWFSA), con reintento y aumento de haz. Dado un ancho de haz *beam* inicial, la decodificación es reiniciada si no se encuentra ningún camino. En cada reintento el ancho de haz es aumentado en un factor $rFactor$ y el número máximo de reintentos es igual a $rMax$.

Input:
A, a `NdWFSA` (Non-deterministic Weighted Finite-State Automaton)
$X = (x_1, \ldots, x_N)$, a sequence of `Symbols`
beam, a real number (the search beam as *logProb*)
$rFactor$, a real number (beam search retry factor)
$rMax$, an integer (max number of beam search retry)

Output:
$T = (t_1, \ldots, t_N)$, a sequence of `Transitions`

Function `decodeNdWFSA` $(A, X, beam, rFactor, rMax) : T$
  do
  | $T \leftarrow$ `decodeNdWFSA` $(A, X, beam)$
  | **if** $|T| > 0$ **then**
  | **return**
  | **end**
  | $beam \leftarrow rFactor \cdot beam$
  | $rMax \leftarrow rMax - 1$
  while $rMax > 0$
end

disten del camino más probable una cantidad superior al valor del haz. La búsqueda en haz reduce la cantidad de caminos a analizar, acelerando la decodificación a costa de no asegurar la obtención del camino óptimo. Si en algún instante el conjunto de posibles caminos es nulo o si, una vez procesada la secuencia completa de símbolos, ninguno de los caminos alcanza un estado final, el algoritmo devuelve una secuencia vacía. Este hecho puede deberse bien a que ciertamente no exista ningún camino (y por tanto el resultado es correcto), bien a que el ancho de haz configurado sea demasiado pequeño (todos los caminos parciales que podrían haber dado lugar a estados finales han sido descartados).

El decodificador de Sautrela (véase Procesador Decoder) cuenta con un mecanismo de reintento automático que, en caso de no obtener ningún camino posible, vuelve a procesar la secuencia de entrada, haciendo uso de un haz aumentado tantas veces como se desee. El Algoritmo 5.3 muestra este mecanismo de reintento con aumento de haz.

5.2. Entrenamiento unificado

De manera análoga al mecanismo unificado de decodificación, Sautrela define un conjunto de interfaces que, dado un autómata de estados finitos ponderado, permite formalizar un mecanismo unificado de entrenamiento o estimación de parámetros a partir de un conjunto de secuencias de observaciones [124, 126]. El hecho de estimar los parámetros internos de un autómata sin tener conocimiento alguno de la naturaleza interna del mismo puede resultar conceptualmente chocante en un primer momento pero, como veremos a continuación, es posible.

Antes de adentrarnos en el entrenamiento unificado, analizaremos de manera general el mecanismo de estimación de ciertos modelos paramétricos, en orden ascendente de complejidad, continuando con el estudio de la estimación de modelos que contienen distribuciones desconocidas, y terminando con el establecimiento del paralelismo existente entre dichos modelos y la interfaz WFSA. Por último, definiremos el mecanismo de entrenamiento de WFSAs.

5.2.1. Estimación de parámetros mediante esperanzas y estadísticos suficientes

Dado un modelo paramétrico descrito por el conjunto $\theta = \{\theta_1, \ldots, \theta_S\}$ de parámetros y el conjunto $X = \{x(1), \ldots, x(N)\}$ de muestras (observaciones) aleatorias, independientes e idénticamente distribuidas (condicionadas a θ), se denominan *estadísticos suficientes* al conjunto de funciones $SS(X) = \{T_1(X), \ldots, T_S(X)\}$ que tienen la propiedad de suficiencia con respecto al modelo y su vector de parámetros [66, 115, 211]. En otras palabras, dado un conjunto de muestras X, los estadísticos suficientes contienen toda la información necesaria para estimar el vector de parámetros θ, y ningún otro estadístico que pueda ser calculado sobre el mismo conjunto de muestras proporciona información adicional alguna.

Los métodos de estimación de cualquier modelo paramétrico se basan justamente en la estimación de los estadísticos suficientes del conjunto de muestras de entrenamiento y la posterior estimación de los parámetros a partir de dichos estadísticos. Los estadísticos suficientes no dependen del criterio de estimación o función optimizada, ya que representan de manera completa al conjunto de muestras. Es por tanto posible combinar los estadísticos suficientes con criterios tales como máxima verosimilitud o máximo a posteriori (ML o MAP, por sus siglas en inglés).

A continuación se analizan cinco ejemplos de modelos paramétricos en orden de complejidad ascendente, definiendo para cada uno de ellos estadísticos suficientes y las fórmulas de estimación de parámetros correspondientes.

Distribución categórica

Sea λ_θ un modelo representado por la familia de distribuciones categóricas sobre K posibles categorías y con vector de pesos $\omega = [\omega_1, \ldots, \omega_K]$ [193]. El espacio muestral de la distribución puede ser definido mediante un sencillo conjunto de índices $x \in \{1, \ldots, K\}$, pero para facilitar la notación, lo definiremos mediante una variable K-dimensional binaria de suma unidad (i.e. $x_k \in \{0, 1\}$ y $\sum_{k=1}^{K} x_k = 1$). La Figura

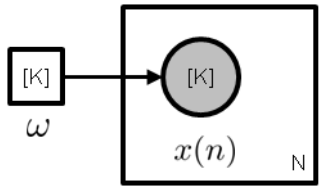


Figura 5.1 Modelo categórico no-Bayesiano expresado en notación *plate*. Los cuadrados menores representan parámetros; los círculos representan variables aleatorias. Las formas sombreadas representan valores observables. Los valores entre corchetes representan las dimensiones de los parámetros o variables. Los rectángulos grandes ("platos", de ahí el nombre de la notación) representan repeticiones de parámetros o variables.

5.1 contiene una representación gráfica de este modelo haciendo uso de la notación denominada *plate*[10] [43, 206]. El conjunto de parámetros $\theta = \{\omega\}$ queda definido por el vector de pesos ω que representa la probabilidad previa de cada componente

$$\omega_k = p\left(x_k = 1\right) \tag{5.1}$$

y la probabilidad marginal del conjunto de muestras $X = \{x\left(1\right), \ldots, x\left(N\right)\}$ viene dada por

$$p\left(X|\theta\right) = \prod_{n=1}^{N} \prod_{k=1}^{K} \omega_k^{x_k} \tag{5.2}$$

En base al criterio de factorización de Neyman-Fisher [66], la función

$$T^{\omega} \quad = \quad T^{\omega}\left(X\right) \quad = \quad \sum_{n=1}^{N} x\left(n\right) \tag{5.3}$$

es un estadístico suficiente. Este estadístico representa toda la información que un conjunto de muestras X posee sobre el vector de pesos ω. La estimación de máxima verosimilitud del conjunto de parámetros

$$\theta_{\text{ML}} \quad = \quad \omega_{\text{ML}} \quad = \quad \arg\max_{\theta} p\left(X|\theta\right) \tag{5.4}$$

viene dada por

$$\omega_{\text{ML}} \quad = \quad \frac{1}{\sum_{k=1}^{K} T_k^{\omega}} T^{\omega} \tag{5.5}$$

Distribución normal multivariante

Sea λ_θ un modelo representado por la familia de distribuciones normales multivariantes $\mathcal{N}\left(\mu, \Sigma\right)$ con vector de medias μ y matriz de covarianza Σ desconocidas, como se muestra en la Figura 5.2. El conjunto de parámetros queda definido por $\theta = \{\mu, \Sigma\}$ y

[10] La notación *Plate* es una notación extendida pero no demasiado estandarizada. En la presente memoria, todos los componentes utilizados en las figuras han sido descritos en su primera aparición.

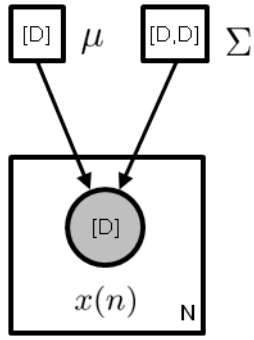


Figura 5.2 Modelo Gaussiano no-Bayesiano expresado en notación *plate*.

la probabilidad marginal del conjunto de muestras $X = \{x(1), \ldots, x(N)\}$ viene dada por

$$p(X|\theta) = \prod_{n=1}^{N} \mathcal{N}(x(n)|\mu, \Sigma) \tag{5.6}$$

y después de alguna manipulación se obtiene

$$p(X|\theta) = |2\pi\Sigma|^{-\frac{N}{2}} \exp\left\{\mu^T\Sigma^{-1}\sum_{n=1}^{N} x(n) - \frac{1}{2}tr\left(\Sigma^{-1}\sum_{n=1}^{N} x(n)\cdot x(n)^T\right) - \frac{N}{2}\mu^T\Sigma^{-1}\mu\right\} \tag{5.7}$$

donde $tr(\cdot)$ denota la traza. En base al criterio de factorización de Neyman-Fisher, las funciones

$$T^0 = T^0(X) = \sum_{n=1}^{N} 1 \tag{5.8}$$

$$T^1 = T^1(X) = \sum_{n=1}^{N} x(n) \tag{5.9}$$

$$T^2 = T^2(X) = \sum_{n=1}^{N} x(n)\cdot x(n)^T \tag{5.10}$$

son estadísticos suficientes. Estos estadísticos representan toda la información que un conjunto de muestras $X = \{x(1), \ldots, x(N)\}$ posee sobre el vector de parámetros $\theta = \{\mu, \boldsymbol{\sigma^2}\}$. La estimación de máxima verosimilitud del conjunto de parámetros

$$\theta_{\text{ML}} = \{\mu_{\text{ML}}, \Sigma_{\text{ML}}\} = \arg\max_{\theta} p(X|\theta) \tag{5.11}$$

viene dada por

$$\mu_{\text{ML}} = \frac{1}{T^0} T^1 \tag{5.12}$$

$$\Sigma_{\text{ML}} = \frac{1}{T^0} T^2 - \mu_{\text{ML}}\cdot\mu_{\text{ML}}^T \tag{5.13}$$

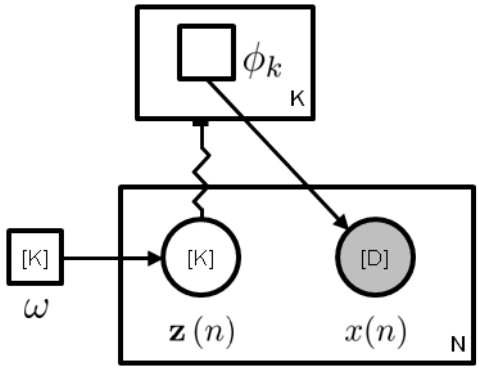


Figura 5.3 Modelo de mezcla no-Bayesiano expresado en notación *plate*. La línea ondulada que termina en una barra transversal indica un interruptor: la variable categórica z selecciona cuál de entre las K distribuciones es responsable de generar la muestra x.

Mezcla de distribuciones

Sea λ_θ un modelo representado por la familia de distribuciones compuestas por la mezcla de K distribuciones $\{\lambda_{\phi_k} \mid k{\in}[1,K]\}$ con vector de pesos $\omega = \{\omega_1, \ldots, \omega_K\}$ y conjuntos de parámetros ϕ_k para cada una de las componentes, como se muestra en la Figura 5.3. El modelo λ_θ es en realidad la composición de una distribución categórica λ_ω con vector de parámetros ω sobre el conjunto $\{\lambda_{\phi_k} \mid k{\in}[1,K]\}$ de K distribuciones donde ω_k representa la esperanza previa de la k-ésima categoría. El conjunto completo de parámetros queda definido por $\theta = \{\omega, \Phi\} = \{\omega_k, \phi_k \mid k{\in}[1,K]\}$ y la probabilidad marginal de un conjunto de muestras $X = \{x(1), \ldots, x(N)\}$ viene dada por

$$p(X|\theta) = \prod_{n=1}^{N} \sum_{k=1}^{K} \omega_k p(x(n)|\phi_k) \tag{5.14}$$

Cada observación x corresponde únicamente a una de las componentes λ_{ϕ_k}. Sin embargo, esa información de pertenencia permanece oculta, ya que la variable categórica $z \sim \lambda_\omega$ no es observable. Al tratarse de una variable latente (no observable), la distribución marginal debe considerar la posible aportación de cada una de las componentes (el sumatorio sobre todas las posibles componentes de la Ecuación 5.14), lo que provoca que no exista una solución en forma cerrada a la estimación de máxima verosimilitud de un modelo de mezcla de distribuciones.

Supongamos que sí conociéramos los valores del conjunto completo de datos $\{X, Z\}$, donde

$$Z = \{z(1), \ldots, z(N)\} \tag{5.15}$$

representa el conjunto de valores de la variable latente z. En tal caso, la probabilidad del conjunto completo de datos puede expresarse como

$$p(X, Z|\theta) = \prod_{n=1}^{N} \prod_{k=1}^{K} \omega_k^{z_k(n)} p(x(n)|\phi_k)^{z_k(n)} \tag{5.16}$$

y, ahora sí, existiría una solución en forma cerrada a la estimación de máxima verosimilitud de los parámetros. Recordemos, sin embargo, que la variable latente z es, por naturaleza, no-observable y que, por tanto, solo tenemos acceso al conjunto incompleto de datos X.

Aunque, a primera vista, el análisis previo de la probabilidad del conjunto completo de datos $\{X, Z\}$ pudiera parecer estéril, es la base del método de re-estimación de esperanza-maximización (EM, por sus siglas en inglés) [52, 110, 24, 199] descrito en el Algoritmo 5.4. El algoritmo EM (en su versión ML) garantiza el aumento de la probabilidad marginal en cada iteración de re-estimación:

$$p\left(X|\theta_{\text{EM-ML}}^{new}\right) \geq p\left(X|\theta^{old}\right) \tag{5.17}$$

Sin embargo, no garantiza la obtención de los parámetros de máxima verosimilitud.

La mecánica del algoritmo EM es más sencilla de lo que pudiera parecer a primera vista. Aunque el conjunto de variables latentes Z no es observable, sí que podemos estimar su esperanza posterior dado un conjunto de muestras X y el modelo λ_θ:

$$E = \mathbb{E}_{X,\theta}\left[Z\right] \tag{5.21}$$

Para ello, deberemos estimar la esperanza $E = \{e\left(1\right), \ldots, e\left(N\right)\}$ del conjunto de variables latentes $Z = \{z\left(1\right), \ldots, z\left(N\right)\}$, donde

$$e\left(n\right) = \mathbb{E}_{X,\theta}\left[z\left(n\right)\right] \tag{5.22}$$

denota la esperanza de la variable latente $z\left(n\right)$. Una vez estimada la esperanza del conjunto completo de datos $\{X, E\}$, se estima el conjunto de parámetros que maximice la distribución compuesta $p\left(X, E|\theta\right)$. Sin embargo, y dado que la esperanza del conjunto completo de datos depende de los parámetros actuales del modelo, llegamos a un algoritmo iterativo de re-estimación que alterna dos pasos: el paso-E, en el que se estima la esperanza del conjunto completo de datos en base al conjunto actual de datos y parámetros

$$E = \mathbb{E}_{X,\theta^{old}}\left[Z\right] \tag{5.23}$$

y el paso-M, en el que se obtiene el conjunto de parámetros que maximiza la probabilidad de la esperanza del conjunto completo de datos

$$\theta_{\text{EM-ML}}^{new} = \arg\max_{\theta} p\left(X, E|\theta\right) \tag{5.24}$$

Un aspecto interesante del algoritmo EM es, sin duda, que las esperanzas de las variables latentes pasan a ser parte del conjunto completo de datos, como si de un observable más se tratara.

Retomando el estudio de la mezcla de distribuciones, la esperanza $e\left(n\right) = \left[e_1\left(n\right), \ldots, e_K\left(n\right)\right]^T$ de la variable latente $z\left(n\right) = \left[z_1\left(n\right), \ldots, z_K\left(n\right)\right]^T$ vendrá dada por

$$e_k\left(n\right) = \mathbb{E}_{X,\theta}\left[z_k\left(n\right)\right] = \frac{\omega_k p\left(x\left(n\right)|\phi_k\right)}{\sum_{j=1}^{K} \omega_j p\left(x\left(n\right)|\phi_j\right)} \tag{5.25}$$

también denominado posterior, probabilidad de pertenencia o responsabilidad de la k-ésima componente. Se tiene además que la esperanza e es una variable K-dimensional

Algoritmo 5.4 Algoritmo EM-ML de esperanza-maximización para máxima verosimilitud.

1. **Inicialización**. Selección de un conjunto inicial de parámetros θ^{old}.

2. **Paso-E (Esperanza**). Estimar la esperanza del logaritmo de la verosimilitud del conjunto completo de datos $\{X, Z\}$, con respecto a la distribución condicional de las variables latentes Z dado el conjunto X de muestras y el conjunto θ^{old} de parámetros actuales. Dicha esperanza viene dada por

$$Q\left(\theta, \theta^{old}\right) = \mathbb{E}_{Z|X,\theta^{old}}\left[\log L\left(\theta; X\right)\right] = \sum_{Z \in \mathbf{Z}} p\left(Z|X, \theta^{old}\right) \log p\left(X, Z|\theta\right) \quad (5.18)$$

3. **Paso-M (Maximización)**. Estimar los parámetros que maximizan la esperanza del logaritmo de la verosimilitud

$$\theta^{new}_{\text{EM-ML}} = \arg\max_{\theta} Q\left(\theta, \theta^{old}\right) \quad (5.19)$$

4. **Convergencia**. Comprobar la convergencia (bien de la probabilidad marginal o del conjunto de parámetros). Si no se cumple el criterio de convergencia, entonces

$$\theta^{old} \leftarrow \theta^{new}_{\text{EM-ML}} \quad (5.20)$$

   y saltar al paso 2.

Nota: El algoritmo EM puede ser utilizado también para obtener una re-estimación MAP (Maximum-A-Posteriori) de los parámetros del modelo: basta con modificar el paso 3 de maximización, donde la función a maximizar será $Q\left(\theta, \theta^{old}\right) + \log p\left(\theta\right)$, siendo $p\left(\theta\right)$ la distribución a priori de los parámetros.

real de suma unidad y valores acotados al rango $[0,1]$ (i.e. $e_k \in [0,1]$ y $\sum_{k=1}^{K} e_k = 1$). En otras palabras, pasamos de una variable latente binaria a una esperanza real. Dado un conjunto de esperanzas $E = \{e(1), \ldots, e(N)\}$, la probabilidad de la esperanza del conjunto completo de datos $\{X, E\}$ viene dada por

$$p(X, E|\theta) = \prod_{k=1}^{K} \omega_k^{\sum_{n=1}^{N} e_k(n)} p(X, E_k|\phi_k) \tag{5.26}$$

donde $E_k = \{e_k(1), \ldots, e_k(N)\}$ representa el conjunto de esperanzas de la k-ésima componente, y

$$p(X, E_k|\phi_k) = \prod_{n=1}^{N} p(x(n)|\phi_k)^{e_k(n)} \tag{5.27}$$

En base al criterio de factorización de Neyman-Fisher, la función

$$T^{\omega} = T^{\omega}(X, E) = \sum_{n=1}^{N} e(n) \tag{5.28}$$

es un estadístico suficiente respecto del vector de pesos ω. La re-estimación EM de máxima verosimilitud del conjunto completo de parámetros

$$\theta_{\text{EM-ML}}^{new} = \left\{ \omega_k^{new}, \phi_k \;\middle|\; k \in [1,K] \right\} = \arg\max_{\theta} p(X, E|\theta) \tag{5.29}$$

viene dada por

$$\omega^{new} = \frac{1}{\sum_{k=1}^{K} T_k^{\omega}} T^{\omega} \tag{5.30}$$

$$\phi_k^{new} = \arg\max_{\phi} p(X, E_k|\phi) \tag{5.31}$$

Comprobamos que para una función categórica de variables latentes, el estadístico suficiente equivale al de la Ecuación 5.3, reemplazando el valor observado por su esperanza, y pudiendo hacer uso de la misma ecuación de estimación.

Afortunadamente, el algoritmo EM no precisa necesariamente la maximización de la probabilidad conjunta $p(X, E|\theta)$. Toda re-estimación que aumente dicha probabilidad implica un aumento de la probabilidad marginal del conjunto de datos [24]:

$$p(X, E|\theta^{new}) \geq p\left(X, E|\theta^{old}\right) \;\rightarrow\; p(X|\theta^{new}) \geq p\left(X|\theta^{old}\right) \tag{5.32}$$

Ello permite relajar la Ecuación 5.31 en aquellos casos en los que no exista una solución en forma cerrada a la maximización

$$\phi_k^{new} = f(X, E_k) \mid p(X, E_k|\phi_k^{new}) \geq p\left(X, E_k|\phi_k^{old}\right) \tag{5.33}$$

Si las distribuciones ϕ_k pertenecen a la familia exponencial [200], que agrupa a muchas de las distribuciones comúnmente utilizadas y cuyos estadísticos suficientes

$T_s(X) = \sum_{n=1}^{N} f_s(x_n)$ pueden expresarse como suma de funciones sobre el conjunto de muestras, puede demostrarse que si

$$SS(X) = \left\{ \sum_{x \in X} f_1(x), \ldots, \sum_{x \in X} f_s(x) \right\} \tag{5.34}$$

son estadísticos suficientes de la distribución marginal $p(X|\phi_k)$, entonces

$$SS(X, E_k) = \left\{ \sum_{x,e \in (X,E_k)} ef_1(x), \ldots, \sum_{x,e \in (X,E_k)} ef_s(x) \right\} \tag{5.35}$$

son estadísticos suficientes del conjunto completo de datos $\{X, E_k\}$ y las fórmulas de estimación $\theta = g(SS)$ basadas en estadísticos suficientes maximizan entonces la distribución conjunta $p(X, E_k|\phi_k)$:

$$g(SS(X)) = \arg\max_{\phi} p(X|\phi) \quad \rightarrow \quad g(SS(X, E)) = \arg\max_{\phi} p(X, E|\phi) \tag{5.36}$$

GMM

Sea λ_θ un modelo representado por la familia de distribuciones compuestas por la mezcla de K distribuciones Gaussianas multivariantes $\{\mathcal{N}(\mu_k, \Sigma_k) \mid {}_{k \in [1,K]}\}$ con vector de pesos $\omega = \{\omega_1, \ldots, \omega_K\}$, vectores de medias μ_k y matrices de covarianza Σ_k desconocidas, como se muestra en la Figura 5.4. El conjunto de parámetros queda definido por $\theta = \{\omega_k, \mu_k, \Sigma_k \mid {}_{k \in [1,K]}\}$ y la probabilidad marginal de un conjunto de muestras $X = \{x(1), \ldots, x(N)\}$ viene dada por

$$p(X|\theta) = \prod_{n=1}^{N} \sum_{k=1}^{K} \omega_k \mathcal{N}(x(n)|\mu_k, \Sigma_k) \tag{5.37}$$

Dado que un GMM no es sino una mezcla de distribuciones normales multivariantes, la esperanza de sus componentes vendrá dada por

$$e_k(n) = \mathbb{E}_{x,\theta}[z_k(n)] = \frac{\omega_k \mathcal{N}(x(n)|\mu_k, \Sigma_k)}{\sum_{j=1}^{K} \omega_j \mathcal{N}(x(n)|\mu_j, \Sigma_j)} \tag{5.38}$$

Además, la distribución normal pertenece a la familia exponencial, por lo que podremos aplicar la Ecuación 5.35 a los estadísticos suficientes de las Ecuaciones 5.8-5.10. En virtud de todo ello se tiene que las funciones

$$T^0 = T^\omega = T^0(X, E) = \sum_{n=1}^{N} e(n) \tag{5.39}$$

$$T_k^1 = T_k^1(X, E) = \sum_{n=1}^{N} e_k(n)\, x(n) \tag{5.40}$$

$$T_k^2 = T_k^2(X, E) = \sum_{n=1}^{N} e_k(n)\, x(n) \cdot x(n)^T \tag{5.41}$$

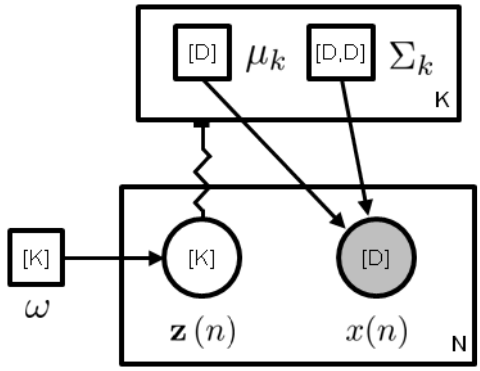


Figura 5.4 Modelo de mezcla de Gaussianas no-Bayesiano expresado en notación *plate*.

son estadísticos suficientes y la re-estimación EM de máxima verosimilitud del conjunto de parámetros

$$\theta_{\text{EM-ML}}^{new} = \left\{ \omega_k^{new}, \mu_k^{new}, \Sigma_k^{new} \;\middle|\; k \in [1,K] \right\} = \arg\max_{\theta} p\left(X, E|\theta\right) \tag{5.42}$$

viene dada por

$$\omega^{new} = \frac{1}{\sum_{k=1}^{K} T_k^0} T^0 \tag{5.43}$$

$$\mu_k^{new} = \frac{1}{T_k^0} T_k^1 \tag{5.44}$$

$$\Sigma_k^{new} = \frac{1}{T_k^0} T_k^2 - \mu_k^{new} \cdot \left(\mu_k^{new}\right)^T \tag{5.45}$$

HMM

Sea λ_θ un HMM, representado por la familia de distribuciones compuestas por K distribuciones $\{ \lambda_{\phi_k} \mid k \in [1,K] \}$. Sea $X = (x(1), \ldots, x(n))$ una secuencia de muestras. Como ocurría en el modelo de mezcla de distribuciones, cada observación x corresponde únicamente a una de las componentes λ_{ϕ_k}, pero esta información de pertenencia permanece oculta, y es representada mediante una variable z latente K-dimensional binaria de suma unidad.

A diferencia del modelo de mezcla, el HMM establece que la distribución de probabilidad de la variable latente correspondiente a una muestra $x(n)$ dependerá de su valor anterior a través de la distribución condicional $p(z(n)|z(n-1))$ (véase Figura 5.5). Para representar dicha dependencia, definiremos la variable matricial latente

$$Z(n) = z(n)\; z(n-1)^T \tag{5.46}$$

que tendrá la naturaleza de variable binaria de dimensión $K \times K$ y suma unidad (i.e. $Z_{i,j}(n) \in \{0,1\}$ y $\sum_{i=1}^{K} \sum_{j=1}^{K} Z_{i,j}(n) = 1$). Para la primera muestra, para la cual

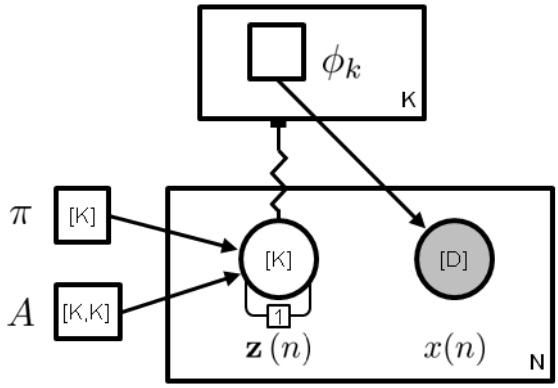


Figura 5.5 HMM no-Bayesiano expresado en notación *plate*. La notación sobre la variable latente z representa el orden de dependencia secuencial de dicha variable. Para un HMM (orden 1) se tiene $p(\mathbf{z}(n)|\mathbf{z}(n-1))$; para un orden genérico o, se tendría $p(\mathbf{z}(n)|\mathbf{z}(n-1),\mathbf{z}(n-2),\ldots,\mathbf{z}(n-o))$

no existe observación previa, y por simplificar la notación, distribuiremos los valores de la variable latente $z(1)$ a lo largo de la diagonal de la variable latente matricial

$$Z_{i,j}(1) = \begin{cases} z_k(1), & \text{si } i = j \\ 0, & \text{en caso contrario} \end{cases} \tag{5.47}$$

La matriz $Z(n)$ contiene también el valor de las variables latentes $z_k(n)$, ya que se tiene

$$z(n) = \left[\sum_{j=1}^{K} Z_{1j}(n), \ldots, \sum_{j=1}^{K} Z_{Kj}(n)\right]^T = Z(n) \cdot \mathbb{1} \tag{5.48}$$

donde $\mathbb{1} = [1,\ldots,1]^T$.

El conjunto de parámetros de un HMM $\theta = \{\pi, A, \phi_k \mid {}_{k\in[1,K]}\}$ se compone de un vector π de probabilidades iniciales, una matriz A de probabilidades de transición[11] y el conjunto de parámetros $\Phi = \{\phi_k \mid {}_{k\in[1,K]}\}$ correspondiente a las K distribuciones de emisión. Nótese que tanto el vector de probabilidades π como cada una de las columnas de la matriz $A = [A_1,\ldots,A_K]$ representan los vectores de parámetros de $K+1$ distribuciones categóricas, por lo que resulta más claro representar el conjunto de parámetros como

$$\theta = \left\{\pi, A_k, \phi_k \;\middle|\; {}_{k\in[1,K]}\right\} \tag{5.49}$$

Las correspondientes variables latentes categóricas pueden expresarse como

[11] Según esta notación, el valor A_{ij} de la matriz de transición representa la probabilidad de transitar del j-ésimo estado al i-ésimo estado. Esta disposición ha sido escogida para facilitar el uso de la notación basada en vectores columna.

$$z^{\pi}(n) = \begin{cases} diag(Z(1)), & \text{si } n = 1 \\ [0,\dots,0]^T, & \text{en caso contrario} \end{cases} \tag{5.50}$$

$$z^{A_k}(n) = Z_k(n),\ n > 1 \tag{5.51}$$

donde $diag(\cdot)$ representa el vector columna compuesto por los elementos de la diagonal de una matriz, y $Z_k(n)$ representa la k-ésima columna de la variable matricial latente $Z(n) = [Z_1(n),\dots,Z_K(n)]$. Sus esperanzas vendrán dadas por

$$\mathbb{E}_{X,\theta}[z^{\pi}(n)] = \begin{cases} diag(E(1)), & \text{si } n = 1 \\ [0,\dots,0]^T, & \text{en caso contrario} \end{cases} \tag{5.52}$$

$$\mathbb{E}_{X,\theta}\left[z^{A}(n)\right] = E_k(n), \qquad n > 1 \tag{5.53}$$

donde $E_k(n)$ representa la k-ésima columna de la esperanza[12] de la variable matricial latente, $E(n) = [E_1(n),\dots,E_K(n)] = \mathbb{E}_{X,\theta}[Z(n)]$.

Dadas las esperanzas de las variables latentes z^{π}y z_k^A, y en base a las Ecuaciones 5.28 y 5.30, se tiene que las funciones

$$T^{\pi} = T^{\pi}(X,E) = diag(E(1)) \tag{5.54}$$

$$T_k^A = T_k^A(X,E) = \sum_{n=2}^{N} E_k(n) \tag{5.55}$$

son estadísticos suficientes con respecto a los vectores de parámetros π y A_k, y la re-estimación EM de máxima verosimilitud del conjunto de parámetros

$$\theta_{\text{EM-ML}}^{new} = \left\{ \pi, A, \phi_k \;\middle|\; k\in[1,K] \right\} = \arg\max_{\theta} p(X,E|\theta) \tag{5.56}$$

viene dada por

$$\pi^{new} = \frac{1}{\sum_{j=1}^{K} T_k^{\pi}} T^{\pi} \tag{5.57}$$

$$A_k^{new} = \frac{1}{\sum_{j=1}^{K} T_{jk}^{A}} T_k^{A} \tag{5.58}$$

$$\phi_k^{new} = \arg\max_{\phi} p(X,\mathcal{E}_k|\phi) \tag{5.59}$$

donde $\mathcal{E}(n) = \mathbb{E}_{X,\theta}[z(n)] = E(n)\cdot\mathbb{1}$ denota la esperanza de la variable latente $z(n)$ asociada al conjunto de distribuciones Φ , y $\mathcal{E}_k = \{\mathcal{E}_k(1),\dots,\mathcal{E}_k(N)\}$ denota el conjunto de k-ésimas componentes.

[12]Debido a la complejidad del proceso de estimación de la esperanza de la variable latente, no es posible aportar una solución en forma cerrada como se ha hecho en los casos anteriores. Existen diferentes variantes de dicha estimación, pero en general pueden ser formalizadas, bien mediante la aplicación del algoritmo conocido como *forward-backward* [145] o *Baum-Welch* [23], que estima globalmente la esperanza de las variables latentes, bien mediante la aplicación del algoritmo conocido como *Viterbi* [188], que acota la estimación de las esperanzas a la secuencia más probable de valores latentes. Llegados a este punto, supondremos que tenemos acceso a dicha estimación, dejando para más adelante el estudio de los diversos métodos de obtención.

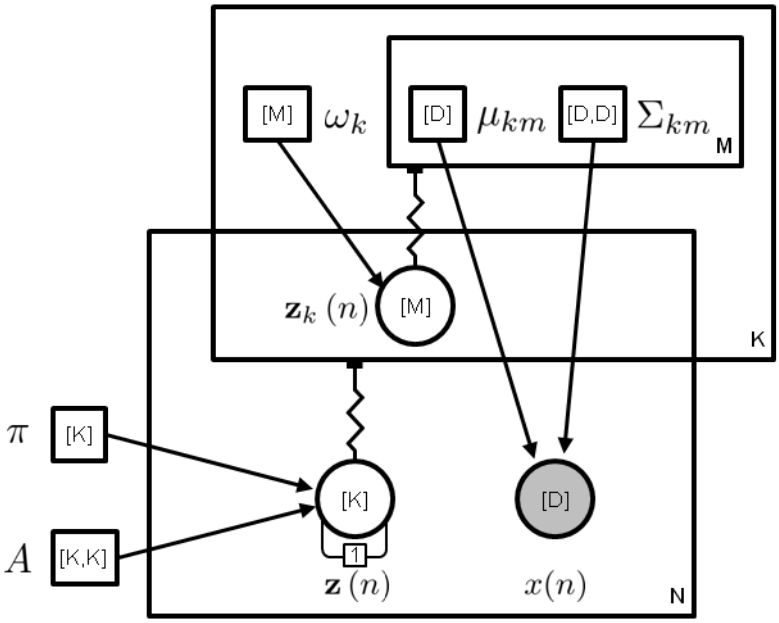


Figura 5.6 HMM no-Bayesiano con distribuciones de mezclas de Gaussianas, expresado en notación *plate*.

Análogamente al apartado anterior, al no haber establecido cuáles son las distribuciones de probabilidad λ_{ϕ_k} que gobiernan el HMM, se ha definido un mecanismo genérico o unificado de re-estimación de HMMs donde, dado un conjunto concreto de distribuciones $\Phi = \{\, \phi_k \mid k \in [1,K] \}$, cada una de ellas deberá maximizar la distribución conjunta $p\,(X, \mathcal{E}_k | \phi_k)$.

HMM-GMM

Sea λ_θ un HMM compuesto por K distribuciones λ_{ϕ_k}, cada una de ellas compuesta a su vez por una mezcla λ_{ω_k} de M distribuciones $\lambda_{\psi_{km}}$ normales multivariantes como se muestra en la Figura 5.6, y también conocido como HMM-GMM o HMM continuo. El conjunto completo de parámetros viene dado por $\theta = \{\, \pi, A_k, \omega_k, \mu_{km}, \Sigma_{km} \mid k \in [1,K], m \in [1,M] \,\}$. Como se ha visto en la Ecuación 5.59, cada una de las distribuciones λ_{ϕ_k} deberá maximizar la distribución conjunta $p\,(X, \mathcal{E}_k | \phi_k)$. La esperanza de la variable latente z^{ω_k} asociada a cada mezcla λ_{ω_k}

$$\epsilon_k = \mathbb{E}_{X,\theta}\left[z^{\omega_k}\,(n) \right] \tag{5.60}$$

vendrá dada por el vector $\epsilon_k = \left[\epsilon_{k1}\,(n), \ldots, \epsilon_{kM}\,(n) \right]^T$, donde

$$\epsilon_{km}\,(n) = \mathcal{E}_k\,(n) \cdot \frac{\omega_{km} \mathcal{N}\left(x\,(n) \,|\mu_{km}, \Sigma_{km} \right)}{\sum_{j=1}^{K} \omega_{kj} \mathcal{N}\left(x\,(n) \,|\mu_{kj}, \Sigma_{kj} \right)} \tag{5.61}$$

representa la responsabilidad posterior de cada distribución normal. En base a las ecuaciones 5.39-5.45 se tiene que

$$T_k^0 = T_k^\omega = T_k^0(X, \mathcal{E}_k) = \sum_{n=1}^{N} \epsilon_k(n) \tag{5.62}$$

$$T_{km}^1 = T_{km}^1(X, \mathcal{E}_k) = \sum_{n=1}^{N} \epsilon_{km}(n)\, x(n) \tag{5.63}$$

$$T_{km}^2 = T_{km}^2(X, \mathcal{E}_k) = \sum_{n=1}^{N} \epsilon_{km}(n)\, x(n) \cdot x(n)^T \tag{5.64}$$

son estadísticos suficientes respecto al conjunto de parámetros $\{\omega_k, \mu_{km}, \Sigma_{km} \mid k \in [1,K], m \in [1,M]\}$ y la re-estimación EM de máxima verosimilitud viene dada por

$$\omega_k^{new} = \frac{1}{\sum_{k=1}^{K} T_{km}^0} T_k^0 \tag{5.65}$$

$$\mu_{km}^{new} = \frac{1}{T_{km}^0} T_{km}^1 \tag{5.66}$$

$$\Sigma_{km}^{new} = \frac{1}{T_{km}^0} T_{km}^2 - \mu_{km}^{new} \cdot (\mu_{km}^{new})^T \tag{5.67}$$

5.2.2. Distribuciones de probabilidad delegables

Diremos que un modelo λ_ϕ representado por la distribución de probabilidad $p(x|\phi)$ es *ML-delegable* cuando, sin necesidad de conocer la naturaleza interna que gobierna dicha distribución, sí que tenemos acceso tanto al valor de la probabilidad marginal de las muestras, $p(x|\phi)$, como a una función o estimador de máxima verosimilitud $\alpha_{\text{ML}}\left(X, E^\phi, \phi\right)$ que, dado un conjunto de muestras X y la esperanza o responsabilidad E^ϕ de la distribución ϕ, obtiene una nueva estimación de los parámetros internos de la distribución

$$\phi^{new} = \alpha_{\text{ML}}\left(X, E_x, \phi^{old}\right)$$

que maximizan la distribución conjunta

$$\phi^{new} = \arg\max_{\phi} p\left(X, E^\phi|\phi\right) \tag{5.68}$$

o en su versión más relajada, garantiza un incremento de dicha probabilidad

$$p\left(X, E^\phi|\phi^{new}\right) \geq p\left(X, E^\phi|\phi^{old}\right)$$

De manera análoga, es posible definir una distribución *MAP-delegable*: basta con modificar la maximización de la Ecuación 5.68, donde la función a maximizar será $p\left(X, E^\phi|\phi\right) \cdot p(\phi)$, siendo $p(\phi)$ la distribución a priori del conjunto de parámetros.

5.2.3. Estimación de modelos con distribuciones desconocidas mediante delegación

El entrenamiento de modelos paramétricos λ_θ, donde el conjunto de parámetros $\theta = \{\rho_j, \phi_k \mid {}_{j\in[1,J],\, k\in[1,K]}\}$ está formado por J parámetros ρ_j y K conjuntos de parámetros ϕ_k pertenecientes a distribuciones desconocidas λ_{ϕ_k}, ya ha sido abordado en dos de los casos analizados en la Sección 5.2.1: mezcla de distribuciones y HMM. En ambos casos, ha sido posible definir un método genérico de re-estimación de los parámetros conocidos mediante el algoritmo EM, independientemente de la naturaleza de las distribuciones λ_{ϕ_k}.

En el paso-E, y únicamente en base a los parámetros conocidos del modelo $\rho^{old} = \{\rho_j^{old} \mid {}_{j\in[1,J]}\}$ y las probabilidades marginales $p(x|\phi_k)$ de las distribuciones desconocidas $\phi = \{\phi_k \mid {}_{k\in[1,K]}\}$, se estimaban las esperanzas $E = \{e(1), \ldots, e(N)\}$ de las variables latentes $Z = \{z(1), \ldots, z(N)\}$ y los estadísticos suficientes $SS\rho(X, E)$ del conjunto de parámetros ρ. En el paso-M, se aplicaban las fórmulas de re-estimación $\rho^{new} = f(SS\rho(X, E))$ basadas en estadísticos que maximizan la probabilidad del conjunto completo de datos $p(X, E)$ y se derivaba un proceso de optimización de cada una de las distribuciones ϕ_k desconocidas frente al conjunto completo de datos $\{X, E^{\phi_k}\}$, donde E^{ϕ_k} corresponde a la esperanza de la variable categórica asociada a la distribución ϕ_k.

Si el conjunto $\phi = \{\phi_k \mid {}_{k\in[1,K]}\}$ de distribuciones desconocidas fuese delegable, estaríamos entonces en condiciones de formalizar un re-estimador $\beta(X, \theta)$ del conjunto completo de parámetros. Nótese que una distribución λ_{ϕ_k} podría a su vez contener distribuciones desconocidas $\lambda_{\psi_{km}}$, por lo que el estimador α_k debería delegar en otros estimadores α_{km} la re-estimación de los modelos $\lambda_{\psi_{km}}$ y así sucesivamente. Debemos constatar, sin embargo, que el mecanismo no puede ser aplicado de manera recursiva, ya que, mientras que los estimadores delegados $\alpha_k\left(X, E^{\phi_k}, \phi_k\right)$ optimizan la probabilidad del conjunto $\{X, E^{\phi_k}\}$, el estimador $\beta(X, \theta)$, tal cual ha sido definido, optimiza la probabilidad del conjunto de muestras $\{X\}$. Es aquí donde los estadísticos suficientes muestran su potencial, ya que las fórmulas de estimación pueden ser escritas en base a ellos, independientemente de si el conjunto de observaciones está o no condicionado a una esperanza de ocurrencia. El Algoritmo 5.5 formaliza los pasos a seguir para re-estimar un modelo compuesto por distribuciones delegables mediante el método de esperanza-maximización-delegación.

5.2.4. WFSA como distribución de probabilidad con variables latentes y distribuciones desconocidas.

Dado un modelo λ_θ representado por un autómata de estados finitos ponderado (WFSA, por sus siglas en inglés), la interfaz de decodificación descrita en la Sección 5.1 establece que podemos acceder al estado inicial s_{ini} de un autómata, así como consultar las transiciones que parten de un estado. Una transición

$$t = (s_{src}, s_{dst}, x, \rho = p(t)) = (src(t), dst(t), obs(t), p(t)) \tag{5.72}$$

será una tupla compuesta por una estado origen s_{src}, un estado destino s_{dst}, un símbolo de emisión u observación x y una probabilidad de transición ρ, donde las funciones

Algoritmo 5.5 Algoritmo EMD-ML de esperanza-maximización-delegación para máxima verosimilitud. Dado un modelo $\theta = \{\rho_j\,,\,\phi_k \mid {}_{j\in[1,J],\,k\in[1,K]}\}$ que contiene un conjunto $\phi = \{\phi_k \mid {}_{k\in[1,K]}\}$ de distribuciones ML-delegables, un conjunto de muestras X y la esperanza E^θ relativa al propio modelo ϕ, formaliza un mecanismo de re-estimación del conjunto completo de parámetros que optimiza la distribución compuesta $p\left(X, E^\theta|, \theta\right)$. Dicha distribución debe ser factorizable.

1. **Inicialización**. Selección del conjunto inicial de parámetros y distribuciones desconocidas.

2. **Paso-E (Esperanza)**. En base a los parámetros actuales del modelo θ^{old}, el conjunto de datos X y la esperanza o responsabilidad E^θ del modelo, estimar la esperanza E^Z de las variables latentes del modelo y los estadísticos suficientes $SS\rho\left(X, E^Z\right)$ del conjunto completo de datos $\left\{X, E^Z\right\}$. Estimar también la esperanza E^{ϕ_k} asociada a cada una de las distribuciones ϕ_k.

3. **Paso-M (Maximización)**. Aplicar las fórmulas de re-estimación del conjunto de parámetros $\left\{\rho_j^{new} \;\middle|\; {}_{j\in[1,J]}\right\}$ basadas en estadísticos suficientes:

$$\rho_{\text{EMD-ML}}^{new} = f\left(SS_\rho\left(X, E^Z\right)\right) = \arg\max_\rho p\left(X, E^Z|\rho\right) \tag{5.69}$$

4. **Paso-D (Delegación)**. Delegar en el conjunto $\left\{\alpha_{\text{ML}}^k\left(X, E^{\phi_k}, \phi_k\right) \;\middle|\; {}_{k\in[1,K]}\right\}$ de estimadores la optimización de las distribuciones desconocidas

$$\phi_k^{new} = \alpha_{\text{ML}}^k\left(X, E^{\phi_k}, \phi_k^{old}\right) = \arg\max_{\phi_k} p\left(X, E^{\phi_k}|\phi_k\right) \tag{5.70}$$

5. **Convergencia**. Comprobar la convergencia de la probabilidad marginal. Si no se cumple el criterio de convergencia, entonces

$$\theta^{old} \leftarrow \theta_{\text{EMD-ML}}^{new} \tag{5.71}$$

   y saltar al paso 2.

Nota: El algoritmo EMD puede también ser utilizado para obtener una re-estimación MAP (Maximum-A-Posteriori), si las distribuciones son MAP-delegables: basta con modificar la maximización del paso 3, donde la función a maximizar será $p\left(X, E^Z|\rho\right) \cdot p\left(\rho\right)$, siendo $p\left(\rho\right)$ la distribución a priori de los parámetros, así como hacer uso en el paso 4 de los estimadores α_{MAP}^k correspondientes.

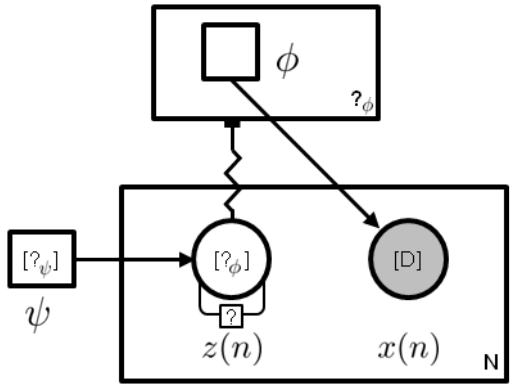


Figura 5.7 Modelo WFSA expresado en notación *plate*, con una variable latente $\mathbf{z}$ desconocida. El orden de dependencia secuencial de la variable $\mathbf{z}$ es desconocido.

$src\,()$, $dst\,()$, $obs\,(\cdot)$ y $p\,(\cdot)$ son aquellas que nos permiten acceder respectivamente a dichos campos. A continuación analizaremos cómo podemos interpretar los elementos de un autómata desde el punto de vista de una distribución de probabilidad con variables latentes y distribuciones desconocidas.

La figura 5.7 muestra una representación gráfica de un WFSA. Podemos suponer que dicha distribución cuenta con una variable latente categórica $\mathbf{z}$ K-dimensional (siendo K desconocido). La distribución de probabilidad de la variable latente correspondiente a una muestra $x\,(n)$ dependerá de sus valores anteriores a través de la distribución condicional regida por el modelo topológico λ_ψ, donde el orden de dependencia es también desconocido. Por ultimo, la variable latente z hace las veces de interruptor, seleccionando cuál de las distribuciones desconocidas ϕ es la responsable de generar la muestra $x\,(n)$ (el número de distribuciones ϕ coincidirá con el tamaño de la variable latente z).

Analicemos antes que nada el orden de dependencia secuencial que establece el modelo λ_ψ a las variables latentes $\mathbf{z}$. Para el caso de un HMM, por ejemplo, ya hemos visto que el orden de dependencia es 1. Para un modelo de N-gramas, el orden será $N-1$. Sea υ el orden de dependencia secuencial por el que se rige el modelo topológico λ_ψ. Se tiene

$$p\left(\mathbf{z}\left(n\right)|\mathbf{z}\left(1\right),\ldots,\mathbf{z}\left(n-1\right),\psi\right)=p\left(\mathbf{z}\left(n\right)|\mathbf{z}\left(n-\upsilon\right),\ldots,\mathbf{z}\left(n-1\right),\psi\right) \tag{5.73}$$

Los estados representan los sucesivos modos o situaciones en las que se encuentra un autómata. Dado que la situación del autómata (su "estado") queda totalmente definida por la secuencia de variables latentes $\left(\mathbf{z}\left(n-\upsilon+1\right),\ldots,\mathbf{z}\left(n\right)\right)$, todo estado s puede entenderse como una representación equivalente de dicha secuencia de variables latentes. Sea $\mathbf{s}=\left[s_1,\ldots,s_{|\mathbf{s}|}\right]^T$ una variable latente categórica (binaria y de suma unidad) L-dimensional ($L=|\mathbf{s}|=K\upsilon$) que representa el estado en el que se encuentra el autómata. Se tiene la equivalencia entre un estado y la secuencia de variables latentes:

$$\mathbf{s}\left(n\right)\equiv\left(\mathbf{z}\left(n-\upsilon+1\right),\ldots,\mathbf{z}\left(n\right)\right) \tag{5.74}$$

Esta equivalencia puede formularse en base a los índices de la variable $\mathbf{s}(n)$ y la secuencia de variables $(\mathbf{z}(n-\upsilon+1),\ldots,\mathbf{z}(n))$. Se tiene

$$i \equiv (k_1,\ldots,k_\upsilon) \iff s_i(n) = \prod_{m=1}^{\upsilon} z_{k_m}(n-\upsilon+m) \tag{5.75}$$

Lógicamente, y en función del orden υ del modelo topológico λ_ψ, podrán existir estados equivalentes para diferentes instantes de una secuencia completa de variables latentes

$$\mathbf{s}(n) = \mathbf{s}(m) \iff \mathbf{z}(n+1-o) = \mathbf{z}(m+1-o) \ , \quad \forall\ 1 \leq o \leq \upsilon \tag{5.76}$$

Por ejemplo, en un HMM ($\upsilon = 1$) la variable de estado queda definido por la última variable latente

$$\mathbf{s}_{HMM}(n) \equiv (\mathbf{z}(n)) \tag{5.77}$$

mientras que en un modelo de 3-gramas ($\upsilon = 2$) la variable de estado representa a las últimas dos variables latentes

$$\mathbf{s}_{3-gram}(n) \equiv (\mathbf{z}(n-1),\mathbf{z}(n)) \tag{5.78}$$

Una transición representa el cambio de estado de un autómata, pasando de un estado origen

$$\mathbf{s}_{src}(n-1) \equiv (\mathbf{z}_{src}(n-\upsilon),\ldots,\mathbf{z}_{src}(n-1)) \tag{5.79}$$

a un estado destino

$$\mathbf{s}_{dst}(n) \equiv (\mathbf{z}_{dst}(n-\upsilon+1),\ldots,\mathbf{z}_{dst}(n)) \tag{5.80}$$

donde la variación de estado es únicamente debida a la n-ésima variable latente, por lo que

$$\mathbf{z}_{dst}(m) = \mathbf{z}_{src}(m) \ , \quad \forall\ n-\upsilon+1 \leq m \leq n-1 \tag{5.81}$$

Un aspecto interesante de diseño de la interfaz WFSA es que permite conceptualizar el autómata en su espacio de estados o variables latentes $\mathbf{s}$, donde la dependencia secuencial pasa a ser de orden 1. La Figura 5.8 muestra una representación gráfica del WFSA con una variable latente de estado $\mathbf{s}$. Puede observarse que el autómata resultante es equivalente al HMM de la Figura 5.5. Si, de manera análoga a lo dispuesto con los HMM, definimos las probabilidades de inicio y transición:

$$\pi = p(\mathbf{s}(0)) \tag{5.82}$$

$$A_k = p(\mathbf{s}(n)\,|s_k(n-1)=1) \tag{5.83}$$

estas son equivalentes al modelo topológico interno

$$\left\{\pi, A_k \,\middle|\, {}_{k\in[1,K]}\right\} \equiv \psi \tag{5.84}$$

Por último, la probabilidad $\rho(n) = p(t(n))$ de una transición $t(n) = (\mathbf{s}_{src}(n-1),\mathbf{s}_{dst}(n),x(n),\rho(n))$ es el producto de la probabilidad condicional de

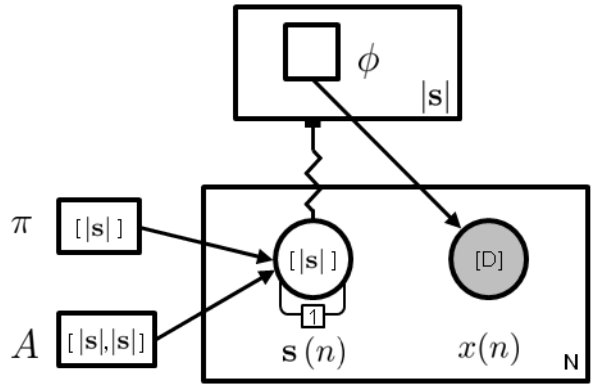


Figura 5.8 Modelo WFSA con una variable latente **s** de estado y expresado en notación *plate*. El orden de dependencia secuencial de la variable **s** es 1.

la variable latente $\mathbf{z}(n)$ y la probabilidad condicional de la observación $x(n)$ dada la distribución ϕ_k seleccionada:

$$\begin{aligned}\rho(n) &= p(x(n), \mathbf{s}_{dst}(n) | \mathbf{s}_{src}(n-1), \pi, A, \phi) \\ &= p(\mathbf{z}(n) | \mathbf{z}(n-\upsilon), \ldots, \mathbf{z}(n-1), \psi) \prod_{k=1}^{K} p(x(n) | \phi_k)^{z_k(n)} \end{aligned} \tag{5.85}$$

Sea Z una la variable tensorial binaria latente de suma unidad, tal que

$$Z_{k_{n-\upsilon},\ldots,k_n}(n) = \prod_{i=n-\upsilon}^{n} z_{k_i}(i) \tag{5.86}$$

La variable $Z(n)$ de dimensionalidad $\upsilon+1$ representa completamente la información latente del modelo. Esta misma información puede ser expresada de forma más sencilla en función de la variable latente **s**. Sea S la variable matricial binaria latente que representa todas las posibles transiciones del autómata

$$S(n) = \mathbf{s}(n)\,\mathbf{s}(n-1)^T \tag{5.87}$$

Entonces ambas variables son equivalentes

$$S(n) \equiv Z(n) \tag{5.88}$$

y en base a la equivalencia de índices de la Ecuación 5.75, su relación viene dada por

$$\left.\begin{array}{l} i \equiv (k_{n-\upsilon}, \ldots, k_{n-1}) \\ j \equiv (k_{n-\upsilon+1}, \ldots, k_n) \end{array}\right\} \rightarrow S_{ji}(n) = Z_{k_{n-\upsilon},\ldots,k_n}(n) \tag{5.89}$$

Los conjuntos completos de datos, así como sus esperanzas son también equivalentes

$$\{X, S\} \equiv \{X, Z\} \tag{5.90}$$

$$\{X, E^{\mathbf{s}} = \mathbb{E}_{X,\theta}[S]\} \equiv \{X, E^{\mathbf{z}} = \mathbb{E}_{X,\theta}[Z]\} \tag{5.91}$$

donde

$$E^{\mathbf{s}}(n) = \mathbb{E}_{X\theta}[S(n)] = p(\mathbf{s}(n)|\mathbf{s}(n-1), X, \theta) \tag{5.92}$$

es la esperanza de ocurrencia o posterior de la transición $\{\mathbf{s}(n-1) \rightarrow \mathbf{s}(n)\}$, y la relación entre ambas esperanzas viene dada por

$$\left.\begin{array}{l} i \equiv (k_{n-v}, \ldots, k_{n-1}) \\ j \equiv (k_{n-v+1}, \ldots, k_n) \end{array}\right\} \rightarrow E^{\mathbf{s}}_{ji}(n) = E^{\mathbf{z}}_{k_{n-v},\ldots,k_n}(n) \tag{5.93}$$

Debido a la equivalencia con un HMM, se tiene que las funciones $\left\{T^{\pi}(X,S), T^{A}_{k}(X,S)\right\}$ de las Ecuaciones 5.54-5.55 son estadísticos suficientes con respecto a los vectores de parámetros π y A_k, y que su re-estimación vendrá dada por las Ecuaciones 5.56-5.59. Si las distribuciones de emisión ϕ son delegables, es posible realizar la re-estimación de todos los parámetros mediante el algoritmo de EMD del apartado anterior, que en su paso-E requiere estimar la esperanza de las variables latentes $E^{\mathbf{s}}$.

Recapitulando, a pesar de no tener acceso al conocimiento de la naturaleza interna del modelo WFSA, su interfaz sí nos da acceso al espacio de estados S, que es equivalente al espacio de variables latentes Z. Al estimar la esperanza $E^{\mathbf{s}}$ de las transiciones, y sin necesidad de conocer la naturaleza interna del modelo, indirectamente estamos estimando la esperanza $E^{\mathbf{z}}$ de las variables latentes, a partir de la cual se formaliza la re-estimación interna del modelo. Este es el fundamento sobre el que se asienta el mecanismo de estimación unificado de Sautrela que se formaliza en el siguiente apartado.

5.2.5. Mecanismo de entrenamiento unificado para WFSAs

El mecanismo de entrenamiento unificado de WFSAs de Sautrela determina que para que un autómata pueda ser entrenado, este deberá ser capaz de estimar sus parámetros a partir de la esperanza de la secuencia de transiciones. Téngase en cuenta que en base a la Ecuación 5.93, dichas esperanzas pueden ser directamente convertidas en componentes de la esperanza de la variable latente interna Z del modelo.

La implementación de este requisito no resulta compleja si se siguen las siguientes directrices:

1. Hacer uso de los estadísticos suficientes. Las fórmulas de re-estimación resultan sencillas y el código resultante es claro y elegante.

2. Hacer uso de distribuciones *auto-delegables*. La implementación de una distribución de probabilidad es *auto-delegable* si ella misma es capaz de estimar los parámetros que maximizan la probabilidad conjunta $p\left(X, E^{\phi_k}|\phi_k\right)$. Las distribuciones auto-delegables permiten crear modelos de complejidad casi ilimitada basada en la composición de distribuciones.

Como solo exigiremos a los modelos la capacidad de estimación de sus parámetros a partir de la esperanza de la secuencia de transiciones, los descargamos de la tediosa tarea de estimar la esperanza de las variables latentes, proceso que será llevado a cabo por el entrenador de Sautrela (véase Procesador Trainer). Podríamos decir que el mecanismo de entrenamiento desacopla el paso-E de esperanza, que queda a cargo del entrenador, del paso-M de maximización, que queda a cargo del modelo.

En base a estos requisitos, la tarea del entrenador se limita a:

1. Calcular las esperanzas de todas las posibles transiciones para cada instante de la secuencia de muestras
2. Transmitir dicha información al modelo, para que este pueda re-estimar sus parámetros.

Los siguientes dos apartados describen estas dos tareas.

Estimación de esperanza de las transiciones

Sea λ_θ un modelo representado por un WFSA que implementa la interfaz de decodificación descrita en la Sección 5.1. Sea $X = (x(1), \ldots, x(N))$ una secuencia de observaciones. De manera similar a lo visto en la Sección 5.1, donde se obtenía la secuencia más probable de transiciones (el camino más probable) dada una secuencia de observaciones, el entrenamiento de un WFSA requiere estimar la probabilidad posterior de todos los posibles caminos.

Dado un autómata determinista, el proceso de estimación de las transiciones resulta sencillo ya que una secuencia X de observaciones dará lugar a una única secuencia de transiciones (ver Algoritmo 5.1) donde la esperanza de cada una de ellas será la unidad.

Para el caso de los autómatas no-deterministas, sin embargo, el proceso de estimación de las transiciones resulta más complejo, ya que una secuencia de muestras dará lugar a infinidad de posibles secuencias de transiciones. El algoritmo conocido como *Baum-Welch* [23] o *forward-backward* [145] permite calcular la esperanza de la variable latente S para cada instante de la secuencia de muestras.

Sean

$$\alpha(n) = \left[\alpha_1(n), \ldots, \alpha_{|\mathbf{s}|}(n)\right]^T \tag{5.94}$$

$$\beta(n) = \left[\beta_1(n), \ldots, \beta_{|\mathbf{s}|}(n)\right]^T \tag{5.95}$$

los vectores de coeficientes de avance (*forward*) y retroceso (*backward*) en el instante n, cuyos valores

$$\alpha_i(n) = p(x(1), \ldots, x(n), s_i(n) = 1) \tag{5.96}$$

$$\beta_i(n) = p(x(n+1), \ldots, x(N), s_i(n) = 1) \tag{5.97}$$

representan, dado un instante n de la secuencia, la suma de las probabilidades de todos los caminos que llegan al i-ésimo estado, y la suma de probabilidades de todos los posibles caminos que parten del i-ésimo estado, respectivamente. Entonces, la probabilidad marginal del conjunto de muestras puede expresarse como

$$p(X) = \alpha(n)^T \cdot \beta(n) \quad , \quad \forall\, 1 \leq n \leq N \tag{5.98}$$

Sea $P(n)$ la matriz[13] de probabilidades del instante n, tal que

$$P_{ij}(n) = p(t(n)) \mid src_j(t) = 1 \wedge dst_i(t) = 1 \tag{5.99}$$

[13] Análogamente a lo dispuesto para los HMM en la Ecuación 5.49, el valor P_{ij} de la matriz de probabilidades de transición representa la probabilidad de transitar del j-ésimo estado al i-ésimo estado. Esta disposición ha sido escogida para facilitar el uso de la notación basada en vectores columna.

Entonces, se tiene la siguiente relación recursiva para el vector de coeficientes de avance

$$\alpha(n) = P(n) \cdot \alpha(n-1) \tag{5.100}$$
$$\alpha(0) = \mathbf{s}_{initial} \tag{5.101}$$

donde debemos recordar que la interfaz WFSA determina que existe un único estado inicial, $\mathbf{s}_{initial}$. De manera análoga, se tiene la siguiente relación recursivas para el vector de coeficientes de retroceso

$$\beta(n) = P(n) \cdot \beta(n+1) \tag{5.102}$$
$$\beta(N) = \left[p_{final}(\mathbf{s}_1), \ldots, p_{final}\left(\mathbf{s}_{|\mathbf{s}|}\right)\right]^T \tag{5.103}$$

donde $p_{final}(\mathbf{s})$ denota la probabilidad de ser final del estado $\mathbf{s}$.

Por último, la esperanza de la variable latente $S(n) = \mathbf{s}(n)\,\mathbf{s}(n-1)^T$ viene dada por

$$E^{\mathbf{s}}(n) = \frac{diag(\beta(n))^T \cdot P(n) \cdot diag(\alpha(n-1))}{p(X)} \tag{5.104}$$

donde la esperanza de cada elemento viene dada por

$$E^{\mathbf{s}}_{ij}(n) = \frac{\beta_i(n) \cdot P_{ij}(n) \cdot \alpha_j(n-1)}{p(X)} \tag{5.105}$$

y haciendo uso de las Ecuaciones 5.98 y 5.100 puede comprobarse que la esperanza es una variable de suma unidad:

$$\sum_{ij} E^{\mathbf{s}}_{ij}(n) = \frac{\sum_{ij} \beta_i(n) \cdot P_{ij}(n) \cdot \alpha_j(n-1)}{p(X)} = \frac{\sum_i \beta_i(n) \cdot \alpha_i(n)}{p(X)} = 1 \tag{5.106}$$

La implementación del estimador de esperanzas debe tener en cuenta dos cuestiones:

1. La interfaz WFSA no da acceso directo al conjunto completo de estados, sino que se accede a ellos indirectamente y bajo demanda, como resultado de las posibles transiciones que parten de otro estado.

2. Debido al orden de dependencia secuencial υ, el espacio de la esperanza $E^{\mathbf{s}}$ es de naturaleza dispersa. Dado que los estados origen y destino de una transición deben cumplir la ecuación 5.81, la esperanza de todas aquellas transiciones "prohibidas" será nula.

Por todo ello la implementación que se mostrará a continuación, en vez de aplicar directamente las ecuaciones matriciales previas, hace uso de una representación dispersa de los estados activos en cada instante [14].

El Algoritmo 5.6 representa la estimación de los vectores de coeficientes α. Para cada muestra de entrada, y en base a las relaciones recursivas de las Ecuaciones 5.100-5.101, obtiene el coeficiente de avance de cada estado activo. Cada vector tiene

[14] Cabría la posibilidad de hacer uso de esas mismas ecuaciones, pero con vectores y matrices dispersos.

Algoritmo 5.6 Algoritmo `forward` para la obtención de los vectores de coeficientes de avance α dado un autómata de estados finitos ponderado no-determinista (NdWFSA) y una secuencia de símbolos (observaciones). Para cada símbolo de entrada x_n, el algoritmo obtiene un mapa F cuyo conjunto de claves representa el conjunto de estados activos, y los valores corresponden a la probabilidad acumulada de todos los caminos que terminan en dicho estado. La función `prune`$(M, beam)$ elimina del mapa todas las entradas correspondientes a estados cuyo logaritmo de probabilidad diste más que el valor *beam* frente al estado más probable. Las probabilidades se suponen dadas en su forma lineal.

Input:
A, a `NdWFSA` (Non-deterministic Weighted Finite-State Automaton)
$X = (x_1, \ldots, x_N)$, a sequence of `Symbols`
beam, a real number

Output:
$Trellis = ((\texttt{NIL}, F_0), (x_1, F_1), \ldots, (x_N, F_N))$, a sequence of tuples, where
x_n, a `Symbol`
$F_n = \{s_n \mapsto \alpha_n\}$, a map where:
s_n, a `State`
α_n, a real number (the probability of all the paths ending on s_n)

Function `forward`$(A,\ X,\ beam)\ :\ T$
$F \leftarrow \{\texttt{getInitState}(A) \mapsto (\{\}, 1)\}$
$Trellis \leftarrow ((\texttt{NIL}, F))$
for $x \in X$ **do**
  $F_{old} \leftarrow F$
  $F \leftarrow \{\mapsto\}$
  for $s_{src} \mapsto p_{old} \in F_{old}$ **do**
    for $(\sim, s_{dst}, \sim, p) \in \texttt{getTrans}(A, s_{src}, x)$ **do**
      if $F[s_{dst}] = \texttt{NIL}$ **then**
        $F[s_{dst}] \leftarrow p_{old} \cdot p$
      else
        $F[s_{dst}] \leftarrow F[s_{dst}] + p_{old} \cdot p$
      end
    end
  end
  $F \leftarrow \texttt{prune}(F,\ beam)$
  if $|M| = 0$ **then**
    $Trellis \leftarrow ()$
    return
  end
  $Trellis \leftarrow Trellis + (F)$
end
end

Algoritmo 5.7 Algoritmo `forwardRetry` para la obtención de los vectores de coeficientes de avance α dado un autómata de estados finitos ponderado no-determinista (NdWFSA) y una secuencia de símbolos (observaciones). Dado un ancho de haz *beam* inicial, el procesamiento es reiniciado si no se encuentra ningún camino. En cada reintento, el ancho de haz es aumentado en un factor $rFactor$. El número máximo de reintentos es igual a $rMax$.

Input:
A, a `NdWFSA` (Non-deterministic Weighted Finite-State Automaton)
$X = (x_1, \ldots, x_N)$, a sequence of `Symbols`
beam, a real number (the search beam as *logProb*)
$rFactor$, a real number (beam search retry factor)
$rMax$, an integer (max number of beam search retries)

Output:
$Trellis = ((\texttt{NIL}, F_0), (x_1, F_1), \ldots, (x_N, F_N))$, a sequence of tuples, where
x_n, a `Symbol`
$F_n = \{s_n \mapsto \alpha_n\}$, a map where:
s_n, a `State`
α_n, a real number (the probability of all the paths ending on s_n)

Function `forwardRetry` $(A, X, beam, rFactor, rMax) : Trellis$
do
  $Trellis \leftarrow$ `forward` $(A, X, beam)$
  if $|Trellis| > 0$ **then**
    return
  end
  $beam \leftarrow rFactor \cdot beam$
  $rMax \leftarrow rMax - 1$
while $rMax > 0$
end

una estructura dispersa implementada mediante un mapa de estados a coeficientes de avance. El resultado del proceso, denominado *Trellis*, contiene la secuencia completa de vectores α. De una manera análoga a la decodificación de un autómata no-determinista (véanse Algoritmos 5.2 y 5.3), el algoritmo de estimación de esperanzas implementa el método de búsqueda en haz, pudiendo descartar los estados muy improbables. El Algoritmo 5.7 implementa además el mecanismo de reintento con aumento de haz. Por lo general, la búsqueda en haz suele ser aplicada solo en los procesos de decodificación, pero Sautrela la incorpora también en la fase de entrenamiento. La búsqueda en haz hace posible la aceleración del proceso de entrenamiento y la reducción del consumo de memoria, y su utilidad aumenta conforme aumentamos el tamaño del autómata a estimar.

El Algoritmo 5.8 muestra la estimación de los vectores de coeficientes β a partir del

Algoritmo 5.8 Algoritmo `backward` para la obtención de la esperanza del conjunto de transiciones dado un autómata de estados finitos ponderado no-determinista (NdWFSA) y un *Trellis* resultante del algoritmo de avance `forward`.

Input:
A, a `NdWFSA` (Non-deterministic Weighted Finite-State Automaton)
$Trellis = ((\texttt{NIL}, F_0), (x_1, F_1), \ldots, (x_N, F_N))$, a sequence of tuples, where
x_n, a `Symbol`
$F_n = \{s_n \mapsto \alpha_n\}$, a map where:
s_n, a `State`
α_n, a real number (the probability of all the paths ending on s_n)

Output:
$E = (s_1, \ldots, s_N)$, a sequence of sets $s_n = \{(t_1, e_1), \ldots, (t_{K_n}, e_{K_n})\}$, where:
t_{k_n}, a `Transition`
$e_{k_n} = p(t_{k_n}|X, A)$, the posterior probability of the transition t_{k_n}

Function `backward` $(A, Trellis) : E$
$E \leftarrow \texttt{NIL}$
for $(x, F) \in reversed(Trellis)$ **do**
  $B \leftarrow \{\mapsto\}$
  if $E = \texttt{NIL}$ **then**
    $E \leftarrow ()$
    $PX \leftarrow 0$
    for $s \mapsto p \in F$ **do**
      $p_{fin} \leftarrow p \cdot \texttt{getFinProb}(A, s)$
      if $p_{fin} > 0$ **then**
        $PX \leftarrow PX + p \cdot p_{fin}$
        $B[s] \leftarrow p \cdot p_{fin}$
      end
    end
  else
    $te \leftarrow \{\}$
    for $s_{src} \mapsto \alpha \in F$ **do**
      $B[s_{src}] \leftarrow 0$
      for $t \in \texttt{getTrans}(A, s_{src}, x)$ **do**
        $(\sim, s_{dst}, \sim, p) \leftarrow t$
        if $B_{old}[s_{dst}] \neq \texttt{NIL}$ **then**
          $B[s_{src}] \leftarrow B[s_{src}] + p \cdot B_{old}[s_{dst}]$
          $te \leftarrow te \cup \{(t, \alpha \cdot p \cdot B_{old}[s_{dst}]/PX)\}$
        end
      end
    end
    $E \leftarrow (te) + E$
  end
  $BB \leftarrow B$
end
end

Algoritmo 5.9 Obtención del conjunto de esperanzas de transiciones dado un WFSA. Si el autómata es determinista, existe a lo sumo un único camino. La búsqueda en haz con reintento puede ser utilizada con los autómatas no-deterministas; el criterio VITERBI utiliza únicamente el camino más probable.

Input:
A, a `WFSA` (Weighted Finite-State Automaton)
$X = (x_1, \ldots, x_N)$, a sequence of `Symbols`
$beam$, a real number (the search beam as *logProb*)
$rFactor$, a real number (beam search retry factor)
$rMax$, an integer (max number of beam search retry)
$criteria \in \{$`VITERBI` , `BAUMBELCH`$\}$, estimation criteria

Output:
$E = (s_1, \ldots, s_N)$, a sequence of sets $s_n = \{(t_1, e_1), \ldots, (t_{K_n}, e_{K_k})\}$, where:
t_{k_n}, a `Transition`
$e_{k_n} = p(t_{k_n}|X, A)$, the posterior of the transition t_{k_n}

Function `expectation` $(A, X, beam, rFactor, rMax, criteria) : E$
$E \leftarrow ()$
if A **instanceof** `DWFSA` **then**
| **for** $t \in$ `decodeDWFSA` (A, X) **do**
| $E \leftarrow E + (\{(t, 1)\})$
| **end**
elseif $criteria =$ `VITERBI` **then**
| **for** $t \in$ `decodeNdWFSA` $(A, X, beam, rFactor, rMax)$
| $E \leftarrow E + (\{(t, 1)\})$
| **end**
else
| $E \leftarrow$ `backward` $(A,$ `forwardRetry` $(A, X, beam, rFactor, rMax))$
end
end

Trellis que contiene los vectores α. El algoritmo aprovecha el proceso de cálculo recursivo de los coeficientes β para calcular las esperanzas de las transiciones. Finalmente la función finalmente retorna una secuencia de conjuntos de esperanzas de transiciones..

Por último, el Algoritmo 5.9 muestra el proceso unificado de estimación de transiciones para todo autómata que implemente la interfaz WFSA. Si se trata de un autómata determinista, existe a lo sumo un único camino posible, cuyas transiciones tendrán una esperanza unidad. Si, por el contrario, se trata de un autómata no determinista, es posible utilizar dos criterios de estimación:

- `BAUMWELCH`. También conocido como *forward-backward* o *soft EM* [23, 145, 199]. Se trata del proceso previamente descrito, en el que se realiza una estimación de la esperanza de las transiciones para cada instante de la secuencia de muestras.

La estimación de las esperanzas puede hacer uso de la búsqueda en haz con reintento.

- `VITERBI`. También conocido como *hard EM* [145, 199] que, a diferencia del método *soft EM*, únicamente considera el camino más probable (aplicando el algoritmo de *Viterbi* [188]), cuyas transiciones tendrán una esperanza unidad. La estimación del camino más probable hace uso de la búsqueda en haz con reintento.

Transmisión de las esperanzas

Una vez formalizado el proceso de estimación de las esperanzas de las transiciones, el mecanismo de transmisión de las mismas resulta sencillo. La interfaz de entrenamiento de un WFSA está compuesta por los cuatro métodos que se muestran en el Listado 5.2. Su cometido es el siguiente:

priorExpectation Este método está íntimamente relacionado con el mecanismo de entrenamiento MAP que implementa Sautrela, pero por ahora nos basta con señalar que inicializa el mecanismo de re-estimación, y podemos usar el valor 0 como argumento de esperanza.

addExpectation Este es el método principal de la interfaz. Permite transmitir al modelo la esperanza de una transición.

addFinalExpectation Dado que la interfaz WFSA determina que los estados deben tener una probabilidad de ser finales, este método permite trasmitir la esperanza que tiene un estado de ser final. Dicho valor resulta de la suma de las esperanzas de todas las transiciones finales que comparten un mismo estado destino.

dumpSuffStats Comunica al modelo que ha finalizado la fase de transmisión de esperanzas y que, por tanto, debe realizar la re-estimación de todos los parámetros internos. La función termina en cuanto finaliza el proceso de re-estimación.

El Algoritmo 5.10 representa el mecanismo unificado de entrenamiento de un WFSA dado un conjunto de secuencias de observaciones, independientemente de la naturaleza interna del modelo. Puede ser utilizado tanto para entrenar un sencillo autómata determinista, como un modelo de n-gramas o un HMM, y las distribuciones de emisión pueden ser discretas o continuas. Como se verá en las siguientes secciones, las posibilidades de este mecanismo unificado son amplificadas por dos elementos:

- La generalización del entrenamiento MAP mediante el método de inicialización-estimación que se analizará en la Sección 5.3.

- La contribución de los modelos de Markov por capas (layered Markov models) que se presentarán en la Sección 5.4 y que permiten, de manera general, el entrenamiento acoplado de WFSAs y, más en concreto, el entrenamiento de un conjunto de WFSAs a partir de secuencias supervisadas de observaciones.

Listado 5.2 Subconjunto de métodos de entrenamiento de la interfaz `WFSA`.

```
public interface WFSA extends Named {
   ...
   public void priorExpectation(double expectation);
   public void addExpectation(Transition t,double expectation);
   public void addFinalExpectation(State s,double expectation);
   public void dumpSuffStats();
   ...
}
```

Algoritmo 5.10 Entrenamiento unificado de un WFSA a partir de conjuntos de observaciones. Todo autómata que implemente la interfaz WFSA podrá ser entrenado.

Input:

A, a `WFSA` (Weighted Finite-State Automaton)

$Xset = \left\{X_1 = \left(x_1, \ldots, x_{N_1}\right), \ldots, X_M = \left(x_1, \ldots, x_{N_M}\right)\right\}$, a set of sequences of `Symbols`

$beam$, a real number (the search beam as *logProb*)

$rFactor$, a real number (beam search retry factor)

$rMax$, an integer (max number of beam search retries)

$criteria \in \{\texttt{VITERBI}, \texttt{BAUMBELCH}\}$, estimation criteria

$prior$, a real number (prior expectation of the WFSA)

Function `train` $(A, X, beam, rFactor, rMax, criteria, prior)$

```
priorExpectation(A, prior)
for X ∈ Xset do
|   for te ∈ expectation(A, X, beam, rFactor, rMax, criteria) do
|       for (t, e) ∈ te do
|           addExpectation(A, t, e)
|       end
|   end
|   for ((∼, s, ∼, ∼), e) ∈ te do
|       addFinalExpectation(A, s, e)
|   end
end
dumpSuffStats(A)
end
```

5.3. Entrenamiento MAP

El criterio de máxima verosimilitud (ML, por sus siglas en inglés) se basa en la hipótesis de que el conjunto de entrenamiento representa suficientemente la fuente que se desea modelar. Sea λ_θ un modelo paramétrico y θ el conjunto de parámetros a estimar a partir del conjunto $X = \{x(1), \ldots, x(N)\}$ de N observaciones independientes. La estimación de máxima verosimilitud (MLE, por sus siglas en inglés) viene dada por

$$\theta_{\text{ML}} = \arg\max_{\theta} \{p(X|\theta)\} \tag{5.107}$$

Sin embargo, la cobertura que ofrece el conjunto de entrenamiento a menudo resulta insuficiente. En unos casos, no se cuenta con suficientes muestras de entrenamiento, como es frecuente en tareas de verificación del locutor, donde se parte de un conjunto muy limitado de señales (a lo sumo unos pocos minutos por hablante). En otros, las condiciones de entrenamiento no coinciden con el dominio de la aplicación, como en el caso de un sistema de transcripción cuyos modelos acústicos han sido entrenados a partir de señales transcritas grabadas en condiciones de laboratorio que distan de las condiciones reales de uso (ruido, eco, reverberación, etc.).

Esta falta de cobertura puede ser compensada mediante información previa sobre los parámetros a estimar. Sea $p(\theta) = f(\theta, \varphi)$ la distribución previa del conjunto de parámetros θ, gobernada por el conjunto de parámetros φ, comúnmente denominado conjunto de hiper-parámetros. Entonces, la estimación según el criterio Maximum A Posteriori (MAP) es aquella que maximiza la distribución posterior $p(\theta|X)$ del conjunto de parámetros θ, y viene dada por

$$\theta_{\text{MAP}} = \arg\max_{\theta} \{p(\theta|X)\} = \arg\max_{\theta} \{p(X|\theta)\, p(\theta)\} \tag{5.108}$$

La estimación MAP, que puede entenderse como una regularización de la estimación ML, permite incorporar conocimiento previo al entrenamiento y previene el sobre-ajuste en casos de insuficiencia de datos. Aunque la elección de la distribución $p(\theta)$ es arbitraria, normalmente suele hacerse uso de la distribución conjugada previa [195]. Como se verá, la razón es bien sencilla. En la teoría de probabilidad Bayesiana, se dice que una distribución posterior $p(\theta|X)$ y una distribución previa $p(\theta)$ son conjugadas si ambas pertenecen a la misma familia. Dadas dos distribuciones conjugadas, la distribución previa $p(\theta)$ es denominada distribución conjugada previa de la verosimilitud $p(X|\theta)$. La elección de la distribución conjugada previa asegura, por tanto, un requerimiento aceptable: que las distribuciones previa $p(\theta)$ y posterior $p(\theta|X)$ pertenezcan a una misma familia.

En la sección anterior, todas las distribuciones de probabilidad han sido de dos tipos: categóricas y normales multivariantes. A continuación se se muestran y analizan sus distribuciones conjugadas previas.

Distribución categórica. Su función de probabilidad viene dada por

$$p(z|\omega) = \prod_{k=1}^{K} \omega_k^{zk} = \exp\left\{z^T \cdot \log \omega\right\} \tag{5.109}$$

donde z es una variable categórica y ω es el vector de pesos o probabilidades previas de cada componente [193]. Su distribución conjugada previa es la distribución de Dirichlet [197], cuya función de densidad de probabilidad viene dada por

$$f\left(\omega|\varphi\right) = \frac{1}{\mathrm{B}\left(\varphi\right)} \prod_{k=1}^{K} \omega_k^{\varphi_k - 1} = \frac{1}{\mathrm{B}\left(\varphi\right)} \exp\left\{\left(\varphi - \mathbb{1}\right)^T \cdot \log \omega\right\} \tag{5.110}$$

donde $\mathbb{1} = \left[1, \ldots, 1\right]^T$ y $\mathrm{B}\left(\cdot\right)$ denota la función multinomial Beta [192].

Distribución normal multivariante. Suele representarse como $\mathcal{N}\left(\mu, \Sigma\right)$ y su función de densidad de probabilidad viene dada por

$$f\left(x|\mu, \Sigma\right) = \left|2\pi\Sigma\right|^{-\frac{1}{2}} \exp\left\{-\frac{1}{2}\left(x - \mu\right)^T \Sigma^{-1} \left(x - \mu\right)\right\} \tag{5.111}$$

Su distribución conjugada previa es la distribución normal-inversa-Wishart [204], que suele representarse como $\mathrm{NIW}\left(\eta, \delta, \Psi, \nu\right)$ y cuya función de densidad de probabilidad viene dada por

$$f\left(\mu, \Sigma|\eta, \delta, \Psi, \nu\right) = \mathcal{N}\left(\mu|\eta, \frac{1}{\delta}\Sigma\right) \cdot \mathcal{W}^{-1}\left(\Sigma|\Psi, \nu\right) \tag{5.112}$$

donde $\mathcal{W}^{-1}\left(\cdot\right)$ denota la función inversa-Wishart [201] que viene dada por la función de densidad de probabilidad

$$f\left(\Sigma|\Psi, \nu\right) = \frac{\left|\Psi\right|^{\frac{\nu}{2}}}{2^{\frac{\nu p}{2}} \Gamma_p\left(\frac{\nu}{2}\right)} \left|\Sigma\right|^{-\frac{\nu - p - 1}{2}} \exp\left\{-\frac{1}{2} tr\left(\Psi\Sigma^{-1}\right)\right\}$$

$\Gamma\left(\cdot\right)$ denota la función multivariante Gamma [202] y p corresponde a la dimensión de las observaciones x. La distribución normal-inversa-Wishart representa el hecho de que el vector de medias sigue una distribución normal con media η y matriz de covarianza $\frac{1}{\delta}\Sigma$

$$\mu \sim \mathcal{N}\left(\eta, \frac{1}{\delta}\Sigma\right) \tag{5.113}$$

mientras que la matriz de covarianza sigue una distribución inversa-Wishart

$$\Sigma \sim \mathcal{W}^{-1}\left(\Psi, \nu\right) \tag{5.114}$$

Cuando la estimación MAP es utilizada para adaptar un modelo θ^{old} estimado sobre un amplio conjunto de muestras X^{old} previo, y que ahora deseamos adaptar a un nuevo escenario representado por un conjunto reducido de muestras X, es común imponer que las modas de las distribuciones previas coincidan con los parámetros anteriores

$$\arg\max_{\theta} \left\{p\left(\theta\right)\right\} = \theta^{old} \tag{5.115}$$

lo que reduce el número de hiper-parámetros. Aplicando la Ecuación 5.115 a las dos distribuciones analizadas, se obtiene

$$\varphi - \mathbb{1} = \lambda\omega^{old} \tag{5.116}$$

$$\eta = \mu^{old} \tag{5.117}$$

$$\frac{\Psi}{\nu - p} = \Sigma^{old} \tag{5.118}$$

Otra forma de reducir aún más el número de hiper-parámetros consiste en imponer que la distribución previa $p(\theta)$ pertenezca a la misma familia que la distribución posterior $p(\theta|X, Z)$, dado el conjunto completo de datos $\{X, Z\}$ y en ausencia de información previa [71]. Para aquellos casos en los que la distribución previa sea el producto de múltiples distribuciones, esta última restricción impone ecuaciones de relación entre los hiper-parámetros de las distintas distribuciones. Aplicado, por ejemplo, a la distribución normal multivariante, cuya distribución previa era el producto de las distribuciones normal $\mathcal{N}\left(\eta, \frac{1}{\delta}\Sigma\right)$ e inversa-Wishart $\mathcal{W}^{-1}(\Psi, \nu)$, se obtiene

$$\nu - p = \delta \tag{5.119}$$

En base a todas estas restricciones, las distribuciones conjugadas previas de las distribuciones categórica y normal multivariante, vienen respectivamente dadas por

$$f(\omega|\lambda) \propto \prod_{k=1}^{K} \omega_k^{\lambda\omega^{old}} \tag{5.120}$$

$$f(\mu, \Sigma|\delta) \propto \mathcal{N}\left(\mu|\mu^{old}, \frac{1}{\delta}\Sigma\right) \cdot \mathcal{W}^{-1}\left(\Sigma|\delta\,\Sigma^{old}, \delta + p\right) \tag{5.121}$$

Como puede verse, ambas distribuciones previas dependen de un único parámetro, λ y δ, respectivamente, que de alguna manera representa la confianza que otorgamos a los parámetros del modelo previo $\lambda_{\theta^{old}}$. Cuanto mayor es la confianza, menor probabilidad asigna la distribución previa a los parámetros que se alejen de los valores previos θ^{old}, y viceversa. En el límite, se tiene

$$\lim_{\lambda,\delta\to\infty} \theta_{\text{MAP}} = \theta^{old} \tag{5.122}$$

$$\lim_{\lambda,\delta\to 0} \theta_{\text{MAP}} = \theta_{\text{ML}} \tag{5.123}$$

Si tratásemos de extrapolar todo lo anterior a una distribución $\theta = \{\omega, \phi_k \mid {}_{k\in[1,K]}\}$ compuesta por una mezcla de distribuciones, deberíamos constatar, en primer lugar, que no existe distribución conjugada previa de la verosimilitud $p(X|\theta)$. Sin embargo, si consideramos el conjunto completo de datos $\{X, Z\}$ que contenga las variables latentes correspondientes, la verosimilitud quedará factorizada, y existirá una distribución conjugada previa. En todos aquellos casos en los que existan variables latentes, la distribución conjugada previa será la correspondiente a la verosimilitud del conjunto completo de datos $\{X, Z\}$.

Por ejemplo, dada una mezcla de distribuciones normales $\theta = \{\omega, \mu_k, \Sigma_k \mid {}_{k\in[1,K]}\}$, la distribución previa $p(\theta)$ será el producto de las distribuciones conjugadas anteriormente analizadas

$$f(\theta|\lambda,\delta) \propto f(\omega|\lambda) \prod_{k=1}^{K} f(\mu_k, \Sigma_k|\delta_k)$$

y el conjunto de hiper-parámetros estará compuesto por el parámetro λ y el vector $\delta = [\delta_1, \ldots, \delta_K]^T$. Si además aplicamos una restricción equivalente a la utilizada en la Ecuación 5.119, se obtiene

$$\delta = \lambda\omega^{old} \tag{5.124}$$

y la distribución conjugada resultante

$$f(\theta|\lambda) \propto \prod_{k=1}^{K} \omega_k^{\lambda\omega^{old}} \mathcal{N}\left(\mu_k|\mu_k^{old}, \frac{1}{\lambda\omega^{old}}\Sigma_k\right) \cdot \mathcal{W}^{-1}\left(\Sigma|\lambda\omega^{old}\Sigma^{old}, \lambda\omega^{old} + p\right) \tag{5.125}$$

queda gobernada por un único parámetro λ, que representa la confianza previa del conjunto θ^{old} de parámetros.

A continuación se analizan las ecuaciones de re-estimación para un HMM con distribuciones de emisión compuestas por mezclas de Gaussianas.

5.3.1. Re-estimación MAP de HMM-GMMs

Sea λ_θ un modelo HMM con K distribuciones de emisión compuestas por mezclas de M Gaussianas multivariantes como el de la Figura 5.6 (también conocido como HMM-GMM o HMM continuo). El conjunto de parámetros viene dado por

$$\theta = \left\{ \pi, A_k, \omega_k, \mu_{km}, \Sigma_{km} \;\middle|\; {}_{k\in[1,K]\,,\ m\in[1,M]} \right\} \tag{5.126}$$

donde π es el vector de probabilidades iniciales, A_k y ω_k son el vector de probabilidades de transición y el vector de pesos del k-ésimo estado, y μ_{km} y Σ_{km} son el vector de medias y la matriz de covarianza de la m-ésima componente del k-ésimo estado. En base a la elección de distribuciones conjugadas previas analizado en el apartado anterior y siguiendo la notación de la Sección 5.2, las fórmulas de re-estimación MAP del conjunto de parámetros (el desarrollo completo[15] puede encontrase en [71]) dada una secuencia

[15] En [71], la re-estimación de las matrices de covarianza viene dada por

$$\begin{aligned}\Sigma_{km}^{new} = \frac{1}{\lambda_k\omega_{km}^{old} + \sum_{n=1}^{N} E_{km}^{\phi}(n)} \Bigg(&\lambda_k\omega_{km}^{old}\Sigma_{km}^{old} + \sum_{n=1}^{N} E_{km}^{\phi}\left(x(n) - \mu_{km}^{new}\right)\left(x(n) - \mu_{km}^{new}\right)^T \\ &+\lambda_k\omega_{km}^{old}\left(\mu_{km}^{old} - \mu_{km}^{new}\right)\left(\mu_{km}^{old} - \mu_{km}^{new}\right)^T\Bigg)\end{aligned}$$

que después de alguna manipulación puede ser expresada como se muestra en la Ecuación 5.131.

de observaciones $X = (x(1), \ldots, x(N))$ vienen dadas por

$$\pi^{new} = \frac{1}{\sum_{j=1}^{K}\left(\rho_j - 1 + diag\left(E^A(1)\right)_j\right)}\left(\rho - \mathbb{1} + diag\left(E^A(1)\right)\right) \quad (5.127)$$

$$A_k^{new} = \frac{1}{\sum_{j=1}^{K}\left(\alpha_{kj} - 1 + \sum_{n=2}^{N} E_{kj}^A(n,t)\right)}\left(\alpha_k - \mathbb{1} + \sum_{n=2}^{N} E_k^A(n)\right) \quad (5.128)$$

$$\omega_k^{new} = \frac{1}{\sum_{l=1}^{M}\left(\lambda_k\omega_{km}^{old} + \sum_{n=1}^{N} E_{kl}^{\phi}(n)\right)}\left(\lambda_k\omega_k^{old} + \sum_{n=1}^{N} E_k^{\phi}(n)\right) \quad (5.129)$$

$$\mu_{km}^{new} = \frac{1}{\lambda_k\omega_{km}^{old} + \sum_{n=1}^{N} E_{km}^{\phi}(n)}\left(\lambda_k\omega_{km}^{old} + \sum_{n=1}^{N} E_{km}^{\phi}(n)\cdot x(n)\right) \quad (5.130)$$

$$\Sigma_{km}^{new} = \frac{1}{\lambda_k\omega_{km}^{old} + \sum_{n=1}^{N} E_{km}^{\phi}(n)}\left(\lambda_k\omega_{km}^{old}\left(\Sigma_{km}^{old} + \mu_{km}^{old}\left(\mu_{km}^{old}\right)^T\right) + \sum_{n=1}^{N} E_{km}^{\phi} x(n)\, x(n)^T\right) - \mu_{km}^{new}\left(\mu_{km}^{new}\right)^T \quad (5.131)$$

donde el conjunto completo de hiper-parámetros viene dado por el conjunto de vectores

$$\varphi = \left\{ \rho, \alpha_k, \lambda \;\middle|\; {}_{k\in[1,K]} \right\} \quad (5.132)$$

5.3.2. Reinterpretación de las ecuaciones MAP

Volviendo Sección 5.2, recordemos que las fórmulas de re-estimación del conjunto de parámetros $\theta = \{\pi, A_k, \omega_k, \mu_{km}, \Sigma_{km} \mid {}_{k\in[1,K]}\,,\ {}_{m\in[1,M]}\}$ de un HMM-GMM en función de los estadísticos suficientes

$$T^{\pi} = T^{\pi}(X,E) = diag(E(1)) \quad (5.133)$$

$$T_k^A = T_k^A(X,E) = \sum_{n=2}^{N} E_k(n) \quad (5.134)$$

$$T_k^0 = T_k^0(X,E) = \sum_{n=1}^{N} \epsilon_k(n) \quad (5.135)$$

$$T_{km}^1 = T_{km}^1(X,E) = \sum_{n=1}^{N} \epsilon_{km}(n)\, x(n) \quad (5.136)$$

$$T_{km}^2 = T_{km}^2(X,E) = \sum_{n=1}^{N} \epsilon_{km}(n)\, x(n)\, x(n)^T \quad (5.137)$$

venían dadas por

$$\pi_k^{new} = \frac{1}{\sum_{j=1}^{K} T_j^{\pi}} T^{\pi} \tag{5.138}$$

$$A_k^{new} = \frac{1}{\sum_{j=1}^{K} T_{kj}^{A}} T_k^{A} \tag{5.139}$$

$$\omega_k^{new} = \frac{1}{\sum_{j=1}^{K} T_{kj}^{0}} T_k^{0} \tag{5.140}$$

$$\mu_{km}^{new} = \frac{1}{T^0} T^1 \tag{5.141}$$

$$\Sigma_{km}^{new} = \frac{1}{T^0} T^2 - \mu^{new} \left(\mu^{new}\right)^T \tag{5.142}$$

donde el vector $\epsilon_k = \mathbb{E}_{X,\theta}\left[z^{\omega_k}\right]$ representaba la esperanza posterior de la variable categórica correspondiente a la mezcla λ_{ω_k}.

Supongamos por un instante que quisiéramos representar la secuencia de observaciones $X^{old} = \left(x\left(1\right), \ldots, x\left(N^{old}\right)\right)$ que dio lugar a la estimación actual del modelo. Dicho conjunto quedaría enteramente definido por sus estadísticos suficientes, que pueden ser expresados en base al conjunto θ^{old} de parámetros previos. Se tiene

$$T^{\pi}\left(X^{old}, E^{old}\right) = diag\left(\mathbb{E}_{X^{old}}\left[Z\left(1\right)\right]\right) = \pi_k^{old} \tag{5.143}$$

$$T_k^{A}\left(X^{old}, E^{old}\right) = \sum_{n=2}^{N^{old}} \mathbb{E}_{X^{old}}\left[Z_k\left(n\right)\right] = V_k^{old}\left(N^{old}\right) \tag{5.144}$$

$$T_k^{0}\left(X^{old}, E^{old}\right) = \sum_{n=1}^{N^{old}} \mathbb{E}_{X^{old}}\left[z^{\omega_k}\left(n\right)\right] = v_k^{old}\left(N^{old}\right)\, \omega_k^{old} \tag{5.145}$$

$$T_{km}^{1}\left(X^{old}, E^{old}\right) = \sum_{n=1}^{N^{old}} \mathbb{E}_{X^{old}}\left[z_m^{\omega_k}\left(n\right)\, x\left(n\right)\right] = v_k^{old}\left(N^{old}\right)\, \omega_{km}^{old}\, \mu_{km}^{old} \tag{5.146}$$

$$\begin{aligned} T_{km}^{2}\left(X^{old}, E^{old}\right) &= \sum_{n=1}^{N^{old}} \mathbb{E}_{X^{old}}\left[z_m^{\omega_k}\left(n\right)\, x\left(n\right) x\left(n\right)^T\right] \\ &= v_k^{old}\left(N^{old}\right)\, \omega_{km}^{old}\left(\Sigma_{km}^{old} + \mu_{km}^{old}\left(\mu_{km}^{old}\right)^T\right) \end{aligned} \tag{5.147}$$

donde, dada la secuencia de observaciones X^{old}, para las esperanzas de las variables latentes $z\left(n\right)$ y $Z\left(n\right)$ se tiene

$$\begin{aligned} \mathcal{E}^{old}\left(n\right) &= \mathbb{E}_{X^{old}}\left[z\left(n\right)\right] = A^{n-1}\pi \\ &= \begin{cases} \pi\,, & \text{si } n = 1 \\ A \cdot \mathcal{E}^{old}\left(n-1\right)\,, & \text{en caso contrario} \end{cases} \end{aligned} \tag{5.148}$$

$$\begin{aligned} E^{old}\left(n\right) &= \mathbb{E}_{X^{old}}\left[Z\left(n\right)\right] = A^{n-1}\pi \cdot \left(A^{n-2}\pi\right)^T \\ &= \begin{cases} A \cdot \pi \cdot \pi^T\,, & \text{si } n = 2 \\ A \cdot E^{old}\left(n\right) \cdot A^T\,, & \text{en caso contrario} \end{cases} \end{aligned} \tag{5.149}$$

y sus esperanzas acumuladas vendrán dadas por

$$\begin{aligned} v^{old}(N) &= \sum_{n=1}^{N} \mathbb{E}_{X^{old}}[z(n)] = \left(\sum_{n=1}^{N} A^{n-1}\right)\pi \\ &= \begin{cases} \pi\,, & \text{si } n=1 \\ \pi + A \cdot v^{old}(n-1)\,, & \text{en caso contrario} \end{cases} \end{aligned} \tag{5.150}$$

$$\begin{aligned} V^{old}(N) &= \sum_{n=2}^{N} \mathbb{E}_{X^{old}}[Z(n)] = \sum_{n=2}^{N} A^{n-1}\pi \cdot \left(A^{n-2}\pi\right)^T \\ &= \begin{cases} A \cdot \pi \cdot \pi^T\,, & \text{si } n=2 \\ A \cdot \pi \cdot \pi^T + A \cdot V^{old}(n-1) \cdot A^T\,, & \text{en caso contrario} \end{cases} \end{aligned} \tag{5.151}$$

Puede comprobarse que los estadísticos de las Ecuaciones 5.143-5.147, aplicados sobre las Ecuaciones 5.138-5.142 de re-estimación dan lugar al conjunto de parámetros θ^{old}.

Consideremos ahora los estadísticos suficientes de la unión de ambos conjuntos de muestras, que al tratarse de distribuciones de la familia exponencial, vendrán dados por la suma de los estadísticos de cada conjunto

$$T_S\left(\left\{X, X^{old}\right\}, \left\{E, E^{old}\right\}\right) = T_S\left(X^{old}, E^{old}\right) + T_S(X, E) \tag{5.152}$$

Una vez aplicados a las Ecuaciones 5.138-5.142, puede comprobarse que las ecuaciones de re-estimación obtenidas son equivalentes a las Ecuaciones 5.127-5.131 si se cumple

$$\rho - \mathbb{1} = \pi^{old} \tag{5.153}$$

$$\alpha_k - \mathbb{1} = V_k^{old}(N) \tag{5.154}$$

$$\lambda = v_k^{old}(N) \tag{5.155}$$

Nótese que los hiper-parámetros ρ y α_k corresponden a vectores de parámetros de $K+1$ distribuciones de Dirichlet, conjugadas previas de las distribuciones categóricas correspondientes a los vectores de parámetros π y A_k, respectivamente. De haberles impuesto a dichas distribuciones previas la condición de que su moda coincidiera con los parámetros previos, de manera análoga a lo dispuesto en la Ecuación 5.116, cada una de las distribuciones previas quedaría gobernada por un único hiper-parámetros

$$\rho - \mathbb{1} = \lambda^{\rho} \cdot \pi^{old} \tag{5.156}$$

$$\alpha_k - \mathbb{1} = \lambda_k^{\alpha} \cdot A_k^{old} \tag{5.157}$$

Aunque no se ha podido comprobar, cabe esperar que si aplicásemos al conjunto de distribuciones previas del HMM la restricción adicional de que la distribución previa $p(\theta)$ pertenezca a la misma familia que la distribución posterior $p(\theta|X, Z)$ dado el conjunto completo de datos $\{X, Z\}$ y en ausencia de información previa, llegaríamos a

ecuaciones que relacionarían entre sí el conjunto reducido de $2K+1$ hiper-parámetros $\varphi = \{ \lambda^{\rho}, \lambda_k^{\alpha}, \lambda_k \mid {}_{k \in [1,K]} \}$, obteniendo finalmente las Ecuaciones 5.153-5.155.

Por tanto, estas ecuaciones restringidas de re-estimación MAP pueden ser reinterpretadas como una estimación de máxima verosimilitud en virtud del conjunto X de observaciones y un conjunto previo de observaciones X^{old} que representa al conjunto previo de parámetros θ^{old}. En tal caso, toda la información de las distribuciones previas de una estimación MAP convencional queda limitada a un único parámetro N^{old}: el cardinal del conjunto previo de datos. Este parámetro representa de manera global la confianza que le otorgamos al conjunto completo de parámetros, y dicha confianza es repartida entre las distribuciones de las que se compone el modelo, en base a sus parámetros previos.

Para el caso de un HMM, la confianza global es repartida primero entre los estados, y la confianza de cada estado es repartida entre el conjunto de distribuciones asociadas a él: las distribuciones de transición y emisión. Las distribuciones de transición y emisión de un estado poco probable obtienen poca confianza y precisarán de pocas muestras representativas para ser adaptadas. De manera análoga, las distribuciones de transición y emisión de un estado muy probable obtienen mucha confianza y precisarán de muchas muestras representativas para ser adaptadas. De manera más general, aquellas distribuciones que correspondan a estados cuya responsabilidad posterior (probabilidad posterior) sea mayor a la previa, serán adaptadas en mayor medida. Además, y de manera general, podemos definir el grado de adaptación global γ, que solo dependerá de la relación entre los cardinales de ambos conjuntos de datos

$$\gamma = \frac{N}{N^{old}} \tag{5.158}$$

Cuanto mayor sea el grado de adaptación γ, mayor adaptación sufrirá el modelo. En el límite, se tendrá

$$\lim_{\gamma \to 0} \theta_{\text{MAP}} = \theta^{old} \tag{5.159}$$

$$\lim_{\gamma \to \infty} \theta_{\text{MAP}} = \theta_{\text{ML}} \tag{5.160}$$

$$\lim_{N, N^{old} \to \infty} \theta_{\text{MAP}} = \theta^{old} = \theta_{\text{ML}} \tag{5.161}$$

Para finalizar, debemos señalar que las estimaciones MAP anteriormente descritas se basan en una única secuencia de observaciones $X = (x(1), \ldots, x(N))$ y la correspondiente secuencia previa $X^{old} = \left(x(1), \ldots, x\left(N^{old}\right)\right)$. Sin embargo, los WFSAs serán entrenados con conjuntos de S secuencias $X = \{X(1), \ldots, X(S)\}$ donde $X(s) = (x(s,1), \ldots, x(s,N_s))$. De forma equivalente, podríamos considerar el conjunto de secuencias previas $X^{old} = \left\{X^{old}(1), \ldots, X^{old}\left(S^{old}\right)\right\}$ compuesto de S^{old} secuencias, donde $X^{old}(s) = \left(x(s,1), \ldots, x\left(s, N_s^{old}\right)\right)$. En tal caso, los estadísticos del

conjunto X vendrán dados por

$$T^{\pi} = T^{\pi}(X,E) = \sum_{s=1}^{S} diag\left(E(s,1)\right) \tag{5.162}$$

$$T_k^A = T_k^A(X,E) = \sum_{s=1}^{S}\sum_{n=2}^{N_s} E_k(s,n) \tag{5.163}$$

$$T_k^0 = T_k^0(X,E) = \sum_{s=1}^{S}\sum_{n=1}^{N_s} \epsilon_k(s,n) \tag{5.164}$$

$$T_{km}^1 = T_{km}^1(X,E) = \sum_{s=1}^{S}\sum_{n=1}^{N_s} \epsilon_{km}(s,n)\; x(s,n) \tag{5.165}$$

$$T_{km}^2 = T_{km}^2(X,E) = \sum_{s=1}^{S}\sum_{n=1}^{N_s} \epsilon_{km}(s,n)\; x(s,n)\, x(s,n)^T \tag{5.166}$$

y los estadísticos del conjunto previo X^{old} vendrán dados por

$$T^{\pi}\left(X^{old},E^{old}\right) = S^{old}\cdot\pi_k^{old} \tag{5.167}$$

$$T_k^A\left(X^{old},E^{old}\right) = \sum_{s=1}^{S^{old}} V_k^{old}\left(N_s^{old}\right) \tag{5.168}$$

$$T_k^0\left(X^{old},E^{old}\right) = \left(\sum_{s=1}^{S^{old}} v_k^{old}\left(N_s^{old}\right)\right)\omega_k^{old} \tag{5.169}$$

$$T_{km}^1\left(X^{old},E^{old}\right) = \left(\sum_{s=1}^{S^{old}} v_k^{old}\left(N_s^{old}\right)\right)\omega_{km}^{old}\,\mu_{km}^{old} \tag{5.170}$$

$$T_{km}^2\left(X^{old},E^{old}\right) = \left(\sum_{s=1}^{S^{old}} v_k^{old}\left(N_s^{old}\right)\right)\omega_{km}^{old}\left(\Sigma_{km}^{old}+\mu_{km}^{old}\left(\mu_{km}^{old}\right)^T\right) \tag{5.171}$$

Nótese que el estadístico relativo a la distribución inicial de la Ecuación 5.143 era independiente del conjunto de observaciones, ya que al considerar una única secuencia, solo contribuía la primera de ellas. Este hecho conllevaba que en la ecuación de re-estimación de dichos parámetros, la aportación MAP fuese independiente del número previo de observaciones, o dicho de otra manera, que no fuera posible regular la esperanza previa de los parámetros del vector de pesos π. Sin embargo, al considerar conjuntos de secuencias, la aportación MAP a la re-estimación del vector de pesos iniciales es variable y su esperanza acumulada previa corresponde al número S de secuencias.

5.3.3. Re-estimación MAP mediante un algoritmo de *Inicialización-Estimación*

Sautrela implementa una estimación MAP unificada que reduce a su mínima expresión el número de hiper-parámetros de las distribuciones previas, pudiendo ser aplicado de forma sencilla sobre cualquier modelo que implemente la interfaz WFSA. Como se ha podido comprobar en el apartado anterior, una re-estimación restringida MAP puede ser interpretada como una re-estimación ML dada la unión del conjunto de secuencias de observaciones y otro conjunto previo representativo de los parámetros actuales del modelo. Las fórmulas de re-estimación pueden obtenerse a partir de una inicialización de los estadísticos suficientes, que dependerá de un único parámetro, el ordinal del conjunto previo de observaciones, que representa la confianza previa o esperanza acumulada que otorgamos al conjunto completo de parámetros del modelo actual. Dicha confianza es transmitida a cada una de las distribuciones de que se compone el modelo, proporcionalmente y en base a su esperanza previa, que podrá ser expresada mediante los parámetros actuales del modelo.

Este algoritmo, que denominaremos *Inicialización-Estimación* (IE), es el utilizado por Sautrela para realizar la re-estimación MAP de cualquier modelo que implemente la interfaz WFSA. El método `priorExpectation` de la interfaz de entrenamiento de Sautrela (véase Figura 5.2) cuenta con un único parámetro que controla la inicialización de los estadísticos suficientes, ofreciendo la posibilidad de realizar una estimación MAP de forma intuitiva y unificada. Se dice que es intuitiva, porque el parámetro de inicialización puede interpretarse como la esperanza previa acumulada o número de observaciones previas de dicho modelo; y se dice que es unificada, porque el algoritmo IE puede ser aplicado a todo modelo que implemente la interfaz de entrenamiento, independientemente de su naturaleza interna. El valor `expectation` controla lo *afilada* que resulta la distribución previa $p(\theta)$ (la fiabilidad de los parámetros previos θ^{old}). Un valor nulo da lugar a una estimación por ML, mientras que según aumentamos su valor, mayor es el aporte de información previa, y en consecuencia, menor es el grado de re-estimación. El entrenador de Sautrela (véase Procesador Trainer) hará uso del método `priorExpectation` para transmitir al modelo la esperanza acumulada o confianza de los parámetros actuales. El modelo deberá transmitir a su vez dicha esperanza acumulada a cada una de las distribuciones internas de las que se compone. Este proceso no resulta complejo si se siguen las siguientes directrices:

1. Hacer uso de los estadísticos suficientes ya, que las fórmulas de inicialización, como se ha podido comprobar para el HMM, resultarán más sencillas.

2. Tener en cuenta que todo modelo secuencial conlleva una función de esperanza previa acumulada de estados y transiciones $v^{old}(N)$ y $V^{old}(N)$ (Ecuaciones 5.150-5.151) que debería tenerse en cuenta a la hora de repartir dicha esperanza acumulada global sobre el conjunto de distribuciones internas. Además, y dado que la interfaz solo permite transmitir la esperanza global $\mathbf{N}^{old} = \sum_{s=1}^{S^{old}} N_s^{old}$ acumulada del modelo (lo que equivale al número total de muestras), no se cuenta con información sobre las longitudes de las secuencias previas (solo se cuenta con la suma). Una posible alternativa es hacer uso de la esperanza de la longitud[16]

[16] Dada la matriz de probabilidades $A_{K+1\times K+1}$ que incorpora la probabilidad final de cada estado

de las secuencias $N_{\mathbb{E}}^{old}$, y por tanto para las ecuaciones 5.167-5.171 se tendría

$$S^{old} = \frac{\mathbf{N}^{old}}{N_{\mathbb{E}}^{old}} \tag{5.172}$$

$$\sum_{s=1}^{S^{old}} V_k^{old}\left(N_s^{old}\right) = S^{old} \cdot V_k^{old}\left(N_{\mathbb{E}}^{old}\right) \tag{5.173}$$

$$\left(\sum_{s=1}^{S^{old}} v_k^{old}\left(N_s^{old}\right)\right) = S^{old} \cdot v_k^{old}\left(N_{\mathbb{E}}^{old}\right) \tag{5.174}$$

Estas últimas ecuaciones pueden simplificarse aún más si suponemos que todos los estados tienen la misma esperanza acumulada, ya que en tal caso se obtiene

$$\sum_{s=1}^{S^{old}} V_k^{old}\left(N_s^{old}\right) = \frac{\left(\mathbf{N}^{old} - S^{old}\right)}{K} \cdot A_k \tag{5.175}$$

$$\sum_{s=1}^{S^{old}} v_k^{old}\left(N_s^{old}\right) = \frac{\mathbf{N}^{old}}{K} \tag{5.176}$$

Todo modelo que implemente la interfaz de entrenamiento es, por tanto, responsable de generar las esperanzas acumuladas previas de cada una de sus distribuciones internas, partiendo de la esperanza acumulada previa del modelo (argumento `expectation` del método `priorExpectation` de la interfaz de entrenamiento de la Figura 5.2) . Dado que la interfaz de entrenamiento es aplicable a cualquier tipo de WFSA, las ecuaciones de inicialización dependerán de la naturaleza propia del modelo (estarán íntimamente relacionadas con las propias ecuaciones de reestimación), si bien las ecuaciones 5.167-5.171 relativas a un HMM podrían servir de ejemplo.

---

mediante un estado final adicional que solo transita sobre sí mismo, la probabilidad previa de las longitudes de secuencia viene dada por el incremento de la esperanza de encontrarse en dicho estado

$$\mathbb{E}\left[z_{K+1}\left(n\right)\right] - \mathbb{E}\left[z_{K+1}\left(n-1\right)\right]$$

donde, si definimos

$$\rho(n) = \mathbb{E}\left[z\left(n\right)\right] - \mathbb{E}\left[z\left(n-1\right)\right] = A^{n-2}\left(A - \mathbb{I}\right)\pi = \begin{cases} \pi\,, & \text{si } n = 1 \\ \left(A - \mathbb{I}\right)\pi\,, & \text{si } n = 2 \\ A \cdot \rho(n-1)\,, & \text{en caso contrario} \end{cases}$$

entonces la probabilidad previa de una longitud n viene dada por $\rho_{K+1}(n)$ y la esperanza de la longitud será

$$\mathbb{E}\left[n\right] = \sum_{n=0}^{\infty} n \cdot \rho_{K+1}(n)$$

que puede estimarse mediante convergencia del valor:

$$\mathbb{E}\left[n\right] \approx \sum_{n=0}^{N \gg 0} n \cdot \rho_{K+1}(n)$$

La estimación MAP es a menudo utilizada para adaptar únicamente un subconjunto de los parámetros de un modelo: es muy típico, por ejemplo, re-estimar las medias de un GMM, manteniendo constantes los pesos y las varianzas. Como ya se ha comentado en la sección 5.2, es el propio modelo el que, a partir de su configuración interna de entrenamiento, decidirá cuáles son los parámetros que deben ser adaptados y cuáles se mantendrán constantes.

5.4. Layered Markov Models

Un modelo de Markov por capas (LMM, por sus siglas en inglés) integra distintos niveles o capas de conocimiento representados por conjuntos de WFSAs, conformando un único autómata que implementa la interfaz WFSA no-determinista [121]. Cada capa, compuesta por un conjunto de WFSAs, modela sus unidades o clases en función de las unidades de la capa inferior, de manera que existe una correspondencia directa entre las observaciones de los modelos de una capa y los modelos de la capa inferior.

Analicemos, por ejemplo, el proceso de decodificación de un reconocedor del habla continua (sistema de dictado). Llamaremos decodificación a la obtención de la secuencia más probable de palabras dada una secuencia de observaciones (que típicamente viene dada en forma de secuencia de vectores de parámetros acústicos). Este proceso implica la combinación de diversos niveles de conocimiento, que a menudo son representados mediante modelos implementados en forma de WFSAs. Típicamente, encontraremos tres capas de conocimiento:

- **Capa 1 - Acústica o subléxica**: Modela cierto conjunto de unidades subléxicas en base a los parámetros acústicos. Esta capa suele denominarse *acústico-fonética*, debido a que los fonemas son una de las elecciones más comunes de entre las unidades subléxicas. Sin embargo, existen múltiples opciones, tales como los difonemas, los trifonemas, las sílabas o las semisílabas. La necesidad de unidades subléxicas viene dada por el hecho de que no es posible modelar acústicamente las unidades léxicas (palabras), ya que normalmente no se cuenta con suficientes muestras de cada una de ellas (de hecho, lo normal es no contar con ninguna realización de muchas de las palabras del vocabulario a reconocer). Por tanto, el uso de unidades subléxicas permite entrenar de manera robusta modelos acústicos de unidades que después pueden utilizarse para formar palabras. En cuanto al tipo de modelos utilizados, los HMM con distribuciones continuas (HMM-GMM u otras variantes) sobre el espacio de parámetros acústicos son una de las elecciones más comunes. Cada modelo subléxico representará una distribución de probabilidad sobre secuencias de parámetros acústicos.

- **Capa 2 - Léxica**: En el caso que nos incumbe, representa el conjunto de unidades a reconocer o decodificar. En virtud de las unidades subléxicas definidas, existirá un conjunto de modelos que representen cada palabra como combinación de dichas unidades. Estos modelos pueden venir dados por sencillas transcripciones canónicas, transcripciones múltiples que den cobertura a variedades dialectales o modelos más complejos que den cobertura a ciertos eventos del habla espontá-

nea[17] tales como interjecciones, dudas, repeticiones, falsos comienzos, etc [149]. En función del nivel de complejidad del modelado léxico, las palabras suelen estar representadas bien por sencillos WFSAs deterministas o bien por HMMs, en ambos casos con distribuciones discretas sobre el conjunto de unidades subléxicas de la capa inferior. Cada modelo léxico representará una distribución de probabilidad sobre secuencias de unidades subléxicas.

- **Capa 3 - Lenguaje**: Dado que la finalidad del proceso es la obtención de la secuencia de palabras más probable, un modelo de lenguaje aporta una distribución previa sobre las posibles secuencias de palabras. A diferencia de las capas anteriores, en este caso existe un único modelo. Existen diversos tipos de modelos que pueden representar un lenguaje. No obstante, y debido a su sencillez y amplia cobertura, los modelos basados en n-gramas son una de las elecciones más comunes. Un modelo de n-gramas puede ser formalizado mediante un WFSA determinista que representará una distribución de probabilidad sobre secuencias de unidades léxicas.

Este esquema puede ser representado mediante un conjunto de capas de conocimiento compuestas de modelos que, a su vez, representan distribuciones de probabilidad sobre secuencias de unidades (modelos) de la capa directamente inferior. Las distribuciones de los modelos de la capa del extremo inferior (capa 1) representarán secuencias de muestras u observaciones acústicas, y en la capa del extremo superior (capa 3 del ejemplo anterior) encontraremos siempre un único modelo. Este mismo esquema podemos encontrarlo de manera explicita o implícita en multitud de situaciones en el contexto de las tecnologías del habla. Analicemos algunas variantes del caso anterior:

- **Decodificación acústico-fonética** (DAF): Se trata de obtener la secuencia más probable de unidades subléxicas. Normalmente se realiza un reconocimiento sin conocimiento fonotáctico previo (a veces denominado de lazo-abierto u *open-loop*), aunque también es posible incorporar al reconocimiento un modelo fonotáctico que aporte información sobre la distribución de probabilidad de las secuencias de fonemas de una lengua, como, por ejemplo, un modelo de n-gramas. Analicemos más en profundidad el primero de los casos. Frecuentemente no se representa explícitamente la distribución previa que se asigna a las secuencias de fonemas, posiblemente debido a que dicha información pasa a ser parte del algoritmo que procesa las observaciones en base al conjunto de modelos acústicos. Sin embargo, dicha información puede ser formalizada, hecho que ayudará a comprender mejor el procesamiento. Una posibilidad sería suponer que toda secuencia es equiprobable, pero dado que la longitud de las secuencias puede ser cualquiera, nos veríamos forzados a utilizar una distribución no normalizada, que podría retornar una probabilidad unitaria para cualquier secuencia. Otra posibilidad sería presuponer una distribución normalizada que suponga que la probabilidad de las secuencias decae con su longitud, que podría ser representada mediante un autómata de un único estado con probabilidad p/K de transitar con cada una de las

[17] Algunos eventos del habla espontánea, como las pausas sonoras, pueden ser modelados también a nivel subléxico.

K unidades subléxicas (la probabilidad de finalizar sería $1 - p$). Ambos modelos dan lugar a diferentes codificaciones, ya que establecen diferentes distribuciones previas a las secuencias de unidades subléxicas. Por lo tanto, en los tres casos (el decodificador en lazo abierto con distribución no normalizada, el decodificador en lazo abierto con distribución normalizada y el decodificador con modelo de n-gramas) el sistema está compuesto de una capa inferior de modelos acústicos y una capa superior con un único modelo fonotáctico[18].

- **Reconocimiento de palabras aisladas**: En este caso, cada secuencia de observaciones corresponde a una única palabra de un vocabulario. En función del tamaño del vocabulario, será o no posible modelar acústicamente (sin necesidad de la capa subléxica) cada una de las palabras. Una vez más, podríamos usar un modelo en el que la probabilidad previa de cada palabra no viniera dada explícitamente. No obstante, es conveniente formalizarla. En cualquier caso, la información previa corresponde a una distribución categórica sobre el conjunto de palabras. Dicha distribución podría ser uniforme o contener información previa específica para cada palabra. Por tanto, el sistema estaría compuesto por dos o tres capas, en función de si las palabras pueden o no ser modeladas acústicamente, respectivamente, y en la capa del extremo superior encontraríamos un único modelo que representa la distribución categórica sobre el conjunto de palabras.

- **Reconocimiento de secuencias regulares**: Nos referimos al caso en el que se desea reconocer una secuencia de palabras pero se establecen ciertas restricciones gramaticales, como por ejemplo en la locución de números[19]. Se trata de un caso muy similar al del sistema de dictado. La única diferencia reside en el modelo de lenguaje, que podría representarse mediante un sencillo WFSA determinista que especifique la gramática de la tarea. Como en el caso anterior, en caso de contar con modelos acústicos del vocabulario, no se precisaría capa subléxica alguna.

- **Reconocimientos de comandos**: Nos referimos al caso en el que se define un conjunto de comandos que pueden ser expresados de manera flexible (no hay una única forma canónica para cada comando), y la finalidad última no es transcribir lo dicho, sino identificar el comando. Podemos comenzar a describir las capas en orden descendente:

  - Primero se precisaría una capa que determine la probabilidad previa de cada comando (una distribución categórica sobre el conjunto de comandos), que podría o no ser uniforme.
  - Debajo tendríamos el conjunto de modelos que representa cada uno de los posibles comandos. Un comando representaría una distribución de probabilidad sobre secuencias de palabras. En función del nivel de flexibilidad que

[18] Las distribuciones utilizadas en el decodificador de lazo-abierto (no normalizada y normalizada) pasan a ser consideradas un modelo fonotáctico más, ya que, al fin y al cabo, ambas representan una distribución de probabilidad sobre las secuencias de fonemas.

[19] En este caso suponemos que la finalidad es obtener la transcripción ortográfica del número, "ciento veintitrés", no la del número 123.

sea preciso, podrían venir representados por sencillos WFSA deterministas o complejos modelos de n-gramas.

- Más abajo aún, tendríamos el conjunto de modelos que representa cada una de las posibles palabras. Suponiendo que no tenemos cobertura para modelar directamente la acústica de cada palabra, una palabra representaría una distribución de probabilidad sobre secuencias de fonemas (u otra unidad subléxica). En función de la cobertura fonética que deseemos dar a los modelos léxicos, estos podrían ser simples WFSA deterministas o complejos HMMs con distribuciones discretas.
- Por último, en la capa del extremo inferior tendríamos el conjunto de modelos acústicos que representa cada una de las posibles unidades subléxicas.

Como puede comprobarse, a pesar de tratarse de una tarea aparentemente más sencilla que la del dictado, cuenta con una capa más de conocimiento.

Todos los casos anteriores se referían únicamente a procesos de decodificación. Sin embargo, aunque no son tan comunes, podemos encontrar situaciones equivalentes en el contexto de procesos de entrenamiento de modelos. El caso más claro se da durante el entrenamiento de modelos acústicos. Típicamente se parte de un conjunto aleatorio de modelos que es re-estimado a partir de un conjunto supervisado de secuencias. El nivel de supervisión puede variar, pero lo más común es contar únicamente con la transcripción ortográfica de frases que han sido leídas por distintos locutores. Este escenario permite plantear diferentes métodos de re-estimación de los modelos acústicos. Los más sencillos tomarán como punto de partida una transcripción canónica de cada secuencia de observaciones, probablemente obtenida mediante alguna herramienta de conversión grafema-fonema, y la posible inclusión de silencios (al menos, al inicio y al final de la secuencia). Dada la transcripción canónica de cada secuencia, se puede:

- Realizar una segmentación automática donde cada secuencia de observaciones es troceada equitativamente en tantos segmentos como fonemas tiene la transcripción canónica de la secuencia. Asignar cada segmento al fonema correspondiente, y utilizarlos para re-estimar los modelos acústicos.

- Obtener la segmentación más probable de la secuencia de observaciones. Asignar cada segmento al fonema correspondiente, y utilizarlos para re-estimar los modelos acústicos.

- Suponer que la secuencia de observaciones ha sido generada por la correspondiente secuencia de modelos acústicos, y re-estimarlos conjuntamente. En este caso, la re-estimación se basará en la esperanza de observación de los modelos, de manera análoga a la estimación de un GMM, donde cada componente es entrenada en base a su esperanza o responsabilidad posterior.

También se puede hacer uso de una transcripción más flexible, por ejemplo permitiendo silencios entre las palabras, o haciendo uso de transcripciones múltiples para cada palabra, o considerando posibles eventos del habla espontánea. Lógicamente, cuanto más

flexible sea lo que ya llamaremos capa de supervisión, mayor complejidad implicará en cuanto a su formalización. Por ejemplo, una supervisión basada en modelos léxicos, donde la transcripción ortográfica representa secuencias de palabras cuyas transcripciones múltiples están representadas en un conjunto de modelos léxicos, podría ser representada mediante dos capas. La superior contendría un único modelo que representa la secuencia de palabras, mientras que la inferior contendría los modelos léxicos. A esto habría que añadirle la capa del extremo inferior que contendrá los modelos acústicos a re-estimar.

Dado un conjunto de capas de supervisión, podrían volver a plantearse las últimas dos opciones mencionadas: utilizar la transcripción para segmentar la secuencia y realizar de manera desacoplada la re-estimación del conjunto de modelos; o entrenar conjuntamente todos los modelos en base a dicha transcripción. Cuanto mejor entrenados estén los modelos acústicos, más fiables serán sus esperanzas posteriores, siendo posible el uso de capas de supervisión más complejas.

Comprobamos, por tanto, que en multitud de ocasiones nos encontramos con configuraciones, tanto de decodificación como de re-estimación, en las que podemos identificar una estructura jerárquica de capas que representan distintos niveles de conocimiento. Cada capa representa un conjunto de distribuciones sobre secuencias de unidades de la capa inferior, salvo la capa del extremo superior, que contiene una única distribución o modelo, y la capa del extremo inferior, que representa un conjunto de distribuciones sobre secuencias de observaciones del sistema global.

Los LMMs ofrecen un formalismo que permite integrar un número ilimitado de capas de conocimiento en un único WFSA no-derminista que implementa la interfaz NdWFSA de Sautrela. Un LMM es equivalente a la composición de dichas capas, de tal manera que, para todos los casos anteriormente analizados, los conjuntos de modelos que intervenían tanto en los procesos de decodificación como de entrenamiento son representados mediante un único autómata. Los complejos procesos de decodificación y entrenamiento frente a conjuntos de modelos pasan a convertirse así en sencillos procesos de decodificación y entrenamiento frente a un único modelo, que representará la composición de todas las capas de conocimiento implicadas.

En cierta medida, un LMM es equivalente a una composición de transductores. Sin embargo, y a diferencia de los *toolkits* del entorno de las tecnologías del habla que hacen uso de transductores a la hora de la decodificación (integrando todas las capas de conocimiento en un único transductor compuesto)[143], un LMM tiene la capacidad adicional de ser re-estimable, transfiriendo la información de re-estimación a cada uno de los modelos que lo componen. Por ejemplo, para el caso anteriormente analizado de entrenamiento de modelos acústicos, los LMMs ofrecen la capacidad de formalizar de manera casi ilimitada los distintos niveles de supervisión que pudieran diseñarse. Por último, y dado que un LMM implementa la interfaz WFSA en su vertiente no determinista, puede ser directamente integrado en los Procesadores Decoder y Trainer, que son los encargados de dirigir las tareas de decodificación y entrenamiento.

5.4.1. Definición formal

Un LMM Φ es un WFSA no determinista compuesto por un vector de N capas

$$\Lambda = [\Lambda_1, \ldots, \Lambda_N] \tag{5.177}$$

donde cada una de las capas representa un nivel de conocimiento y está compuesta por un conjunto de WFSAs Λ_n, a excepción de la última capa, que viene dada por un único modelo

$$\Lambda_n = \begin{cases} \{\, \lambda_{nk} \mid {}_{k\in[1,K_n]} \} \,, & n < N \\ \{\lambda_N\} \,, & n = N \end{cases} \tag{5.178}$$

Los modelos λ_{nk} de una capa representan distribuciones de probabilidad sobre secuencias de un espacio común de observaciones $\mathbf{X}_n$

$$p\left(x\left(1\right), \ldots x\left(T\right) | \lambda_{nk}\right) \,, \quad x\left(t\right) \in \mathbf{X}_n$$

y existe una función inyectiva[20] de correspondencia entre las observaciones de una capa y los modelos de la capa inmediatamente inferior[21]

$$f: \; \mathbf{X}_n \;\; \rightarrow \;\; \Phi_{n-1} \tag{5.179}$$

En consecuencia, el modelo global representa una distribución de probabilidad sobre el espacio de observaciones de la capa del extremo inferior

$$p\left(x\left(1\right), \ldots x\left(T\right) | \Lambda\right) \,, \quad x\left(t\right) \in \mathbf{X}_1$$

Un estado $\mathbf{s}$ de un LMM, en adelante hiper-estado, viene definido por un vector de N estados

$$\mathbf{s} = [s_1, \ldots, s_N] \,, \quad \lambda\left(s_n\right) \in \Lambda_n$$

donde cada estado s_n pertenece a un modelo de la n-ésima capa, y

$$\begin{aligned} \lambda: \; S \;\; &\rightarrow \;\; \bigcup_{n=1}^{N} \Lambda_n \\ s_n \;\; &\rightarrow \;\; \lambda_n \end{aligned} \tag{5.180}$$

denota la función de correspondencia entre estados y modelos. De esta manera, un hiper-estado del LMM representa la combinatoria sobre los estados de los modelos de las capas que conforman el LMM.

[20] No se requiere que la función sea biyectiva; sin embargo, una función no biyectiva indica que algunos modelos de la capa inferior nunca serán utilizados y que, por tanto, deberían haber sido descartados en origen.

[21] Esta función de correspondencia prohibe la existencia de modelos de distribuciones continuas en las capas superiores, ya que de lo contrario el conjunto de modelos de la capa inferior debería ser infinito.

5.4.2. Hiper-transiciones

Una transición $\mathbf{t}$, en adelante hiper-transición, de un LMM Φ, como en cualquier otro autómata, representa un cambio de hiper-estado ante la observación de una muestra x. Sin embargo, y debido a la complejidad de la estructura de un LMM, el mecanismo de transiciones es la clave sobre la que se asienta el formalismo LMM. Un cambio de un hiper-estado origen

$$src\left(\mathbf{t}\right) = \mathbf{s}^{src} = \left[s_1^{src}, \ldots, s_N^{src}\right] \tag{5.181}$$

a otro hiper-estado destino

$$dst\left(\mathbf{t}\right) = \mathbf{s}^{dst} = \left[s_1^{dst}, \ldots, s_N^{dst}\right] \tag{5.182}$$

representa, en el caso general, la combinatoria de diversas transiciones a distintos niveles, por lo que comenzaremos analizando el caso más sencillo, para terminar formalizando el caso general. Para cada uno de ellos, haremos primero un paralelismo frente a un sistema de dictado de tres capas como el que se ha descrito en el apartado anterior, para posteriormente formalizarlo en base al LMM.

Hiper-transiciones de nivel 1

En el caso más sencillo, la hiper-transición solo supondrá un cambio de estado en la capa inferior. Retomemos el caso de un sistema de dictado, en el que teníamos tres capas involucradas. Un estado del LMM representa la combinación de un estado acústico (por ejemplo un estado interno de un HMM), un estado léxico (por ejemplo un estado dentro del modelo de transcripción fonética de una palabra), y un estado de lenguaje (por ejemplo un estado interno que represente la últimas dos palabras en un modelo de 3-gramas). Ante una nueva muestra, si el estado acústico no es final, lo único que ocurrirá será que el modelo acústico transitará a uno o más estados internos distintos.

Para el LMM, las transiciones se limitarán al modelo

$$\lambda\left(s_1^{src}\right) \in \Lambda_1$$

de la capa del extremo inferior. Dada la observación x y el modelo $\lambda\left(s_1^{src}\right)$, existirá un conjunto de transiciones

$$T_1 = trans\left(\lambda\left(s_1^{src}\right), s_1^{src}, x\right) \tag{5.183}$$

que darán lugar a un conjunto $\mathbf{T}\left(\Lambda, \mathbf{s}^{src}, x, 1\right)$ de hiper-transiciones de nivel *1*, donde cada una de las hiper-transiciones $\mathbf{t} \in \mathbf{T}\left(\Lambda, \mathbf{s}^{src}, x, 1\right)$ quedará enteramente representada por la tupla compuesta por el hiper-estado origen $\mathbf{s}^{src}$ y la transición $t_1 \in T_1$ de la capa inferior

$$\mathbf{t} = \left(\mathbf{s}^{src}, t_1\right), \quad t_1 \in trans\left(\lambda\left(s_1^{src}\right), x\right) \tag{5.184}$$

Decimos que la hiper-transición queda enteramente representada por la tupla $\left(\mathbf{s}^{src}, t_1\right)$ porque el resto de componentes (hiper-estado destino, observación y probabilidad) pueden ser inferidos a partir de ella. En concreto, el hiper-estado destino vendrá dado por

$$dst\left(\mathbf{t}\right) = \left[dst\left(t_1\right), s_2^{src}, \ldots, s_N^{src}\right] \tag{5.185}$$

la observación x corresponderá a la observación de la transición de la capa inferior

$$obs\left(\mathbf{t}\right) = obs\left(t_1\right) \tag{5.186}$$

y la probabilidad será igual a la probabilidad de la transición de la capa inferior

$$p\left(\mathbf{t}\right) = p\left(t_1\right) \tag{5.187}$$

Hiper-transiciones de nivel 2

Supongamos ahora que el estado de la capa inferior tiene una probabilidad no nula de ser final. En el sistema de dictado, nos encontraríamos con que el estado del modelo acústico puede ser final, y por tanto debemos considerar la posibilidad de un cambio de fonema. Ello conlleva transitar en el modelo léxico que nos encontremos. Nótese que se deben considerar todas las posibles transiciones del modelo léxico ya que su transición no está condicionada a ninguna observación. Cada una de las posibles transiciones determinará la correspondiente unidad acústica, cuyo modelo acústico deberá partir de su estado inicial y transitar con la observación x, pudiendo dar lugar a uno o más estados destino[22].

En el formalismo LMM, dado un hiper-estado origen cuyo estado de la capa inferior tiene probabilidad no nula de ser final, pueden existir hiper-transiciones de nivel *2* debidas a posibles transiciones en la segunda capa. Dado el modelo $\lambda\left(s_2^{src}\right) \in \Phi_2$ de la segunda capa, debe considerarse el conjunto de posibles transiciones incondicionales (no condicionadas a una observación)

$$T_2 = trans\left(\lambda\left(s_2^{src}\right), s_2^{src}\right) \tag{5.188}$$

Cada una de las transiciones $t_2 \in T_2$ determinará una observación $obs\left(t_2\right)$ que corresponderá con un modelo de la capa directamente inferior (la correspondencia viene dada por la función inyectiva de la Ecuación 5.179)

$$\lambda_1 = f\left(obs\left(t_2\right)\right) \in \Lambda_1 \tag{5.189}$$

y para cada uno de los modelos λ_1, dada la observación x, existirá un conjunto de transiciones que partan de su único estado inicial

$$T_1 = transInit\left(\lambda_1, x\right) \tag{5.190}$$

donde $transInit\left(\lambda, x\right) = trans\left(\lambda, s^{ini}, x\right)$ denota el subconjunto de transiciones que parten del estado inicial s^{ini} del modelo λ dada una observación x.

En definitiva, el encadenamiento $\{t_2 \rightarrow t_1\}$de las transiciones de la capa 2 con las correspondientes en la capa 1 dará lugar a un conjunto de hiper-transiciones de nivel *2*

[22]Normalmente los HMM son formalizados en base a un conjunto de estados de emisión (cada estado representa una distribución sobre el espacio de observaciones) y un conjunto de transiciones para cada uno de los estados. Sin embargo, la interfaz WFSA de Sautrela establece que las emisiones están ligadas a las transiciones. En la Sección 4.7 ya se ha mostrado que es posible implementar dicha interfaz suponiendo el autómata equivalente de la Figura 4.5. En este sentido el conjunto de transiciones que parten del estado inicial de la Figura 4.5-b equivale a la composición de las probabilidades iniciales de los estados y sus emisiones.

$\mathbf{T}(\Lambda, \mathbf{s}^{src}, x, 2)$, donde cada hiper-transición $\mathbf{t} \in \mathbf{T}(\Lambda, \mathbf{s}^{src}, x, 1)$ quedará enteramente representada por la tupla compuesta por el hiper-estado origen $\mathbf{s}^{src}$ y el vector $[t_2, t_1]$ de transiciones

$$\mathbf{t} = (\mathbf{s}^{src}, [t_2, t_1]) \ , \quad t_i \in \begin{cases} trans\left(\lambda\left(s_2^{src}\right), s_2^{src}\right) \ , & i = 2 \\ transInit\left(f\left(obs\left(t_2\right)\right), x\right) \ , & i = 1 \end{cases} \tag{5.191}$$

El hiper-estado destino vendrá dado por

$$dst\left(\mathbf{t}\right) = \left[dst\left(t_1\right), dst\left(t_2\right), s_3^{src}, \ldots, s_N^{src}\right] \tag{5.192}$$

la observación x corresponderá a la observación de la transición de la capa inferior

$$obs\left(\mathbf{t}\right) = obs\left(t_1\right) \tag{5.193}$$

y la probabilidad de la hiper-transición será el producto de la probabilidad final del estado origen de la capa 1 y las probabilidades de transición de las capas 2 y 1, respectivamente[23]:

$$p\left(\mathbf{t}\right) = p_{fin}\left(\lambda\left(s_1^{src}\right), s_1^{src}\right) \cdot p\left(t_2\right) \cdot p\left(t_1\right) \tag{5.194}$$

Hiper-transiciones de nivel *p*

Siguiendo el esquema anterior, resulta posible plantear transiciones de cualquier nivel. En un sistema de dictado, por ejemplo, una transición de nivel 3 correspondería a partir de un estado acústico final, que a su vez corresponde con un estado léxico final (final de palabra), por lo que deberíamos buscar todas las posibles transiciones en el modelo de lenguaje. Para cada nueva palabra, el modelo léxico debería transitar desde su estado inicial a todos los posibles estados destinos y el modelo acústico correspondiente debería estimar todas las posibles transiciones dada la observación x. En un sistema con más de 3 capas, podríamos plantear de igual manera toda la combinatoria de transiciones.

Desde el punto de vista de un LMM, una hiper-transición de nivel p es aquella que surge a partir de la existencia de estados finales consecutivos en el hiper-estado origen, hasta llegar a un nivel p en el que el modelo $\lambda\left(s_p^{src}\right) \in \Lambda_p$ genera un conjunto de transiciones incondicionales

$$T_p = trans\left(\lambda\left(s_p^{src}\right), s_p^{src}\right) \tag{5.195}$$

Cada una de las transiciones $t_p \in T_p$ determinará una observación $obs\left(t_p\right)$ que corresponderá con un modelo de la capa directamente inferior

$$\lambda_{p-1} = f\left(obs\left(t_p\right)\right) \in \Lambda_{p-1} \tag{5.196}$$

[23] Nótese que la probabilidad de la hiper-transición contiene dos aportaciones de nivel 1: por un lado la probabilidad final $p_{fin}\left(\lambda\left(s_1^{src}\right), s_1^{src}\right)$ y por otro lado la probabilidad de transición $p\left(t_1\right)$. Ello es debido a que las transiciones finales no llevan asociada emisión alguna, por lo que la observación $x = obs\left(\mathbf{t}\right) = obs\left(t_1\right)$ es generada por la transición t_1 que surge a raíz de la transición t_2. Nótese también que, para los hiper-estados origen y destino, los modelos de la capa 1, $\lambda\left(s_1^{src}\right)$ y $\lambda\left(s_1^{dst}\right)$, serán, por lo general, diferentes.

y para cada uno de los modelos λ_{p-1}, existirá un conjunto de transiciones incondicionales que partan del único estado inicial del modelo,

$$T_{p-1} = transInit\left(\lambda_{p-1}\right) \tag{5.197}$$

donde $transInit\left(\lambda\right) = trans\left(\lambda, s^{ini}\right)$ denota el conjunto completo de transiciones que parten del estado inicial s^{ini} del modelo λ. Este último paso será repetido hasta que lleguemos al nivel del extremo inferior (nivel 1), en el cual, para cada uno de los modelos $\lambda_1 = f\left(obs\left(t_2\right)\right) \in \Lambda_1$. Ahora sí, dada la observación x, existirá un conjunto de transiciones que partan del único estado inicial del modelo

$$T_1 = transInit\left(\lambda_1, x\right) \tag{5.198}$$

Todos los posibles encadenamientos $\{t_p \rightarrow t_{p-1} \rightarrow \cdots \rightarrow t_1\}$ de transiciones darán lugar a un conjunto $\mathbf{T}\left(\Lambda, \mathbf{s}^{src}, x, p\right)$ de hiper-transiciones de nivel p del LMM, donde cada hiper-transición $\mathbf{t} \in \mathbf{T}\left(\Phi, \mathbf{s}^{src}, x, p\right)$ quedará enteramente representada por la tupla compuesta por el hiper-estado origen $\mathbf{s}^{src}$ y el vector $[t_p, t_{p-1}, \ldots, t_1]$ de transiciones[24]:

$$\mathbf{t} = \left(\mathbf{s}^{src}, [t_p, t_{p-1}, \ldots, t_1]\right) , \quad t_i \in \begin{cases} trans\left(\lambda\left(s_p^{src}\right), s_p^{src}\right) , & i = p \\ transInit\left(f\left(obs\left(t_{i+1}\right)\right)\right) , & 1 < i < p \\ transInit\left(f\left(obs\left(t_2\right)\right), x\right) , & i = 1 \end{cases} \tag{5.199}$$

El hiper-estado destino vendrá dado por

$$dst\left(\mathbf{t}\right) = \left[dst\left(t_1\right), \ldots, dst\left(t_p\right), s_{p+1}^{src}, \ldots, s_N^{src}\right] \tag{5.200}$$

La observación x, como siempre, corresponderá a la observación de la transición de la primera capa

$$obs\left(\mathbf{t}\right) = obs\left(t_1\right) \tag{5.201}$$

y la probabilidad de la hiper-transición será el producto de las probabilidades finales de los estados origen hasta la capa p y las probabilidades del vector de transiciones

$$p\left(\mathbf{t}\right) = \left(\prod_{i=1}^{p-1} p_{fin}\left(\lambda\left(s_i^{src}\right), s_i^{src}\right)\right) \cdot \left(\prod_{i=p}^{1} p\left(t_i\right)\right) \tag{5.202}$$

Conjunto completo de hiper-transiciones condicionadas a una observación

Dado un LMM Λ formado por N capas, un hiper-estado $\mathbf{s}^{src}$ y una observación x, el conjunto completo de hiper-transiciones $\mathbf{T}\left(\Lambda, \mathbf{s}^{src}, x\right)$ viene dado por

$$\mathbf{T}\left(\Lambda, \mathbf{s}^{src}, x\right) = \bigcup_{p=1}^{N} \mathbf{T}\left(\Lambda, \mathbf{s}^{src}, x, p\right) \tag{5.203}$$

[24]Nótese que las transiciones están dispuestas en orden de ocurrencia, ya que es la transición t_p la que determina una observación, a la cual le corresponderá un modelo λ_{p-1} de la capa inferior, el cual realizará una transición t_{p-1} a partir de su estado inicial. La transición t_{p-1} a su vez determinará una observación, a la cual le corresponderá un modelo λ_{p-2} de la capa inferior, etc.

donde, dado que las hiper-transiciones de nivel p requieren que todos los estados origen de capas inferiores tengan una probabilidad final no nula, se tiene

$$\begin{aligned}\mathbf{T}\left(\Lambda,\mathbf{s}^{src},x\right) &= \bigcup_{p=1}^{n}\mathbf{T}\left(\Lambda,\mathbf{s}^{src},x,p\right)\\ n &= \text{mín}\left\{N,k\in[1,N-1]\;\middle|\;p_{fin}\left(\lambda\left(s_k^{src}\right),s_k^{src}\right)=0\right\}\end{aligned}\tag{5.204}$$

Conjunto completo de hiper-transiciones incondicionales

De manera equivalente a lo dispuesto para el conjunto $\mathbf{T}\left(\Lambda,\mathbf{s}^{src},x,p\right)$ de hiper-transiciones de nivel p, es posible definir el conjunto $\mathbf{T}\left(\Lambda,\mathbf{s}^{src},p\right)$ de hiper-transiciones incondicionales de nivel p. Por incondicional nos referimos a que la hiper-transición no está condicionada a ninguna observación x. Lo único a tener en cuenta es que para cada encadenamiento $\{t_p\rightarrow t_{p-1}\rightarrow\cdots\rightarrow t_2\}$ de transiciones, a diferencia de la Ecuación 5.198, se deben considerar todas las posibles transiciones incondicionales de la capa del extremo inferior

$$T_1 = transInit\left(\lambda_1\right)\tag{5.205}$$

Y el conjunto completo de hiper-transiciones incondicionales $trans\left(\Lambda,\mathbf{s}^{src},x\right) = \mathbf{T}\left(\Lambda,\mathbf{s}^{src}\right)$ vendrá dado por

$$\begin{aligned}\mathbf{T}\left(\Lambda,\mathbf{s}^{src}\right) &= \bigcup_{p=1}^{n}\mathbf{T}\left(\Lambda,\mathbf{s}^{src},p\right)\\ n &= \text{mín}\left\{N,k\in[1,N-1]\;\middle|\;p_{fin}\left(\lambda\left(s_k^{src}\right),s_k^{src}\right)=0\right\}\end{aligned}\tag{5.206}$$

donde lógicamente se cumplirá

$$\mathbf{T}\left(\Lambda,\mathbf{s}^{src},x\right)\subset\mathbf{T}\left(\Lambda,\mathbf{s}^{src}\right)\tag{5.207}$$

5.4.3. Hiper-estado inicial

Como todo WFSA, un LMM Λ contará con un único hiper-estado inicial $\mathbf{s}^{ini}$ que podrá transitar a otros hiper-estados del LMM. A diferencia del resto de los estados del LMM, las hiper-transiciones $\mathbf{t}\in\mathbf{T}\left(\Lambda,\mathbf{s}^{ini}\right)$ que parten del hiper-estado inicial $\mathbf{s}^{ini}$ son siempre de nivel N y surgen debido a las posibles transiciones del estado inicial s_N^{ini} del único modelo λ_N de la N-ésima capa

$$\forall\,\mathbf{t}\in\mathbf{T}\left(\Lambda,\mathbf{s}^{ini}\right)\;,\quad|\mathbf{t}|=N\;\wedge\;src\left(t_N\right)=s_N^{ini}\tag{5.208}$$

y sus probabilidades no tienen en cuenta probabilidad final alguna

$$p\left(\mathbf{t}\right)=p\left(t_N\right)\cdot p\left(t_N\right)\cdot\cdots\cdot p\left(t_1\right)\tag{5.209}$$

Es posible evitar esta excepción de procesamiento si consideramos que el hiper-estado inicial viene dado por el vector

$$\mathbf{s}^{ini}=\left[s_1^{ini},\ldots,s_{N-1}^{ini},s_N^{ini}\right]\tag{5.210}$$

donde, para todas las capas excepto la última, los estados iniciales corresponden a un modelo especial $\lambda_{finalOnly}$ compuesto por un único estado con probabilidad unidad de ser final, y sin transición alguna. El estado de la capa final corresponderá al estado inicial del único modelo de dicha capa

$$s_n^{ini} = \begin{cases} iniState\left(\lambda_N\right), & \text{si } n = N \\ iniState\left(\lambda_{finalOnly}\right), & \text{en caso contrario} \end{cases} \tag{5.211}$$

Puede comprobarse que si se generan las transiciones de acuerdo a las Ecuaciones 5.210 y 5.211 (como si el hiper-estado $\mathbf{s}^{ini}$ fuera otro hiper-estado cualquiera), se cumplirán las Ecuaciones 5.208 y 5.209.

5.4.4. Probabilidad final

Todo hiper-estado $\mathbf{s}$ tiene una probabilidad de ser final igual al producto de las probabilidades finales de los estados que lo conforman

$$p_{fin}\left(\Lambda, \mathbf{s}\right) = \prod_{i=1}^{N} p_{fin}\left(\lambda\left(s_i\right), s_i\right) \tag{5.212}$$

y se cumple el hecho de que la suma de las probabilidades de todas las transiciones que parten de un hiper-estado y su probabilidad final es la unidad

$$\left(\sum_{\mathbf{t} \in \mathbf{T}(\Lambda, \mathbf{s})} p\left(\mathbf{t}\right)\right) + p_{fin}\left(\Lambda, \mathbf{s}\right) = 1 \tag{5.213}$$

5.4.5. Interfaz WFSA: Decodificación

Como ya se ha mencionado al inicio de esta sección, un LMM implementará la interfaz WFSA en su variante no-determinista, Nd-WFSA. El no-determinismo surge del hecho de que, incluso si las capas estuvieran compuestas por autómatas deterministas, toda capa salvo la del extremo inferior equivale a un conjunto de variables latentes no observables que implican un comportamiento global no-determinista. Nótese que, como se ha podido comprobar en el análisis de las hiper-transiciones, para las capas superiores deben ser consideradas todas las posibles transiciones, por lo que, de manera general, dado un hiper-estado origen $\mathbf{s}^{src}$ y una muestra x, el cardinal del conjunto de transiciones $\left|\mathbf{T}\left(\Lambda, \mathbf{s}^{src}, x\right)\right|$ podrá ser mayor que 1.

El Listado 5.1 contiene el subconjunto de interfaces que definen la interfaz de decodificación de un WFSA. De entre todos los componentes, algunos merecen especial atención cuando hablamos de LMMs. A continuación se analizan los más relevantes:

- **Símbolos**: Representan las observaciones de un modelo. Dado que las observaciones de un LMM corresponden a las observaciones de la capa del extremo inferior, los símbolos ya vienen dados por dichos modelos (el LMM no tiene que crear dicha abstracción).

- **Alfabeto**: Representa el espacio $\mathbf{X}$ de observación o conjunto de posibles símbolos de un modelo. El alfabeto viene dado, una vez más, por la capa del extremo inferior. No obstante, es preciso que todos los modelos que componen una capa de conocimiento compartan un mismo alfabeto. Además, el LMM establece que la función inyectiva de correspondencia entre observaciones (símbolos) de una capa y modelos de la capa inferior de la Ecuación 5.179 venga dada por los nombres de los símbolos y los modelos

$$\begin{aligned} f: \mathbf{X}_n &\rightarrow \Lambda_{n-1} \\ x &\rightarrow \lambda, \quad name(x) = name(\lambda) \end{aligned} \tag{5.214}$$

  Es decir, para todo símbolo del alfabeto de una capa deberá existir un único modelo en la capa inferior, tal que el nombre del símbolo coincida con el nombre del modelo.

- **Nombre de hiper-transiciones**: El nombre de una transición es un aspecto interesante relacionado con el mecanismo de decodificación de Sautrela. Dada una secuencia de observaciones $X = (x(1), \ldots, x(K))$ y un modelo Λ, la finalidad última del proceso de decodificación es la obtención de la secuencia más probable de transiciones

$$\mathbf{T} = \underset{\mathbf{T}=(\mathbf{t}(1),\ldots,\mathbf{t}(K))}{\arg\max} \left\{ \prod_{k=1}^{K} p(\mathbf{t}(k)) \,\middle|\, \Lambda, X \right\} \tag{5.215}$$

  El Procesador Decoder, una vez obtenida dicha secuencia, delega en las propias transiciones la obtención de una secuencia de cadenas de caracteres que volcará a la salida. Un sistema de dictado debería volcar a la salida una secuencia de palabras, mientras que un decodificador acústico-fonético (DAF) debería volcar una secuencia de fonemas, y un sistema de reconocimiento de comandos debería volcar la categoría del comando locutado[25]. Analicemos los siguientes aspectos:

  - La longitud de la secuencia de salida es menor o igual que la longitud de la secuencia de entrada (secuencia de observaciones acústicas).
  - En los tres casos expuestos, los elementos de la secuencia de salida corresponden al alfabeto de la capa del extremo superior (palabras para el dictado, fonemas para el DAF y categorías de comandos para el tercero). Sin embargo, podríamos diseñar sistemas donde esta regla no se cumpla. Podríamos, por ejemplo, crear un sistema de dictado que contuviera diferentes dominios de lenguaje modelados de manera diferente: una gramática sencilla basada en reglas para locución de números y un modelo de lenguaje genérico basado en n-gramas. Todo ello requeriría una capa de supervisión, que podría determinar si una secuencia de entrada puede corresponder solo a uno de los dominios, o si puede contener secuencias de segmentos que corresponden a

[25] Por ejemplo, si existiese la categoría MOSTRAR_HORA y el locutor dijera "Podrías decirme qué hora es, por favor", la salida del reconocedor no sería la transcripción a palabras, sino la transcripción a categorías, de tal manera que un sistema automático podría realizar alguna acción en respuesta al comando reconocido.

alguno de los dos dominios. En tal caso, seguiríamos interesados en obtener una secuencia de palabras; sin embargo, las palabras ya no corresponderían al alfabeto de la capa del extremo superior, sino al de la capa inmediatamente inferior (la penúltima capa empezando a contar desde abajo). Al crear un LMM (véase Comando LMM), tendremos la posibilidad de determinar cuál es la capa de decodificación (*dLayer*), de tal manera que el nombre de la transición venga dado por el nombre de la transición de la capa de decodificación

$$name\left(\mathbf{t}=\left(\mathbf{s}^{src},\left[t_p,t_{p-1},\ldots,t_1\right]\right)\right)=\begin{cases} name\left(t_{dLayer}\right), & p\geq dLayer \\ \texttt{NIL}, & p<dLayer \end{cases} \tag{5.216}$$

El Procesador Decoder únicamente volcará a la salida la secuencia de nombres no nulos correspondiente a la secuencia óptima de transiciones $\mathbf{T}=\left(\mathbf{t}\left(1\right),\ldots,\mathbf{t}\left(K\right)\right)$, obteniendo así la secuencia de unidades deseada.

5.4.6. Interfaz WFSA: Entrenamiento

Recordemos que el entrenamiento de WFSAs (véase Algoritmo 5.10) consta de tres fases:

1. **Inicialización de las esperanzas previas de los modelos**. Un único parámetro que representa la esperanza acumulada previa del modelo permite implementar una re-estimación MAP. En caso de desear realizar una re-estimación ML, el valor del parámetro será 0 (la esperanza previa acumulada es nula). En cualquier caso, la llamada al método

$$\Lambda.\texttt{priorExpectation}\left(e\right)$$

donde e representa la esperanza previa acumulada del LMM, conllevará la llamada al método

$$\lambda.\texttt{priorExpectation}\left(e_\lambda\right)$$

para todos los modelos de todas las capas, donde e_λ representa la esperanza previa acumulada del modelo λ y surge del reparto proporcional de la esperanza e entre los modelos de cada capa

$$e_{\lambda_i\in\Lambda_i}=p\left(\lambda_i\right)\cdot e \tag{5.217}$$

donde $p\left(\lambda_i\right)$ representa la probabilidad a priori del modelo λ_i, y debe cumplirse

$$\sum_{\lambda_i\in\Lambda_i}p\left(\lambda_i\right)=1 \tag{5.218}$$

Un LMM puede ser configurado para estimar las probabilidades $p\left(\lambda_i\right)$ en base a dos criterios distintos[26]:

[26] Actualmente, Sautrela solo soporta la transmisión de esperanzas previas en base al criterio equiprobable.

- **Equiprobable**. Se presupone que los modelos de cada capa son equiprobables

$$p\left(\lambda_i \in \Lambda_i\right) = \frac{1}{|\Lambda_i|} \tag{5.219}$$

Este sencillo método puede resultar suficiente en la mayoría de los casos, ya que a menudo las diferencias entre las probabilidades previas de las clases de una capa de conocimiento no suelen ser elevadas.

- **Aproximado**. El LMM estima, partiendo de la capa superior y en orden descendente, la probabilidad a priori de cada modelo. Dicha estimación se basa en tomar el modelo de la capa superior y obtener múltiples secuencias aleatorias, hasta que las distribuciones de observaciones converjan[27]. Estas distribuciones determinarán la probabilidad a priori de cada modelo inferior. En la capa inmediatamente inferior se vuelve a realizar la misma operación, y la probabilidad a priori de cada modelo de la capa inferior será la suma ponderada de las probabilidades de los símbolos correspondientes para cada uno de los modelos superiores:

$$p\left(\lambda_i \in \Lambda_i\right) = \begin{cases} 1\,, & i = N \\ \sum_{\lambda_{i+1} \in \Lambda_{i+1}} p\left(f^{-1}\left(\lambda_i\right) | \lambda_{i+1}\right) p\left(\lambda_{i+1}\right)\,, & i < N \end{cases} \tag{5.220}$$

Esta segunda estimación es una aproximación de las probabilidades a priori reales. Sin embargo, podría introducir un retardo considerable si el cardinal de alguno de los conjuntos de modelos fuera excesivamente elevado.

2. **Estimación y transmisión de esperanzas**. Para cada una se las secuencias de entrada se estima la esperanza de las transiciones, que después es transmitida al modelo a través de la interfaz de entrenamiento. La estimación de las esperanzas es similar a la decodificación; de hecho, solo hace uso del subconjunto de interfaces de decodificación. Una vez estimadas las esperanzas de las transiciones, el Procesador Trainer transmitirá dichas esperanzas al LMM a través del método `addExpectation`$(\cdot)$, que a su vez las deberá retransmitir a cada uno de los modelos implicado en la hiper-transición. En un instante de la secuencia de observaciones, dado el conjunto completo de hiper-transiciones y sus correspondientes esperanzas $\{\mathbf{T}, E\}$ podríamos obtener la esperanza de cualquier transición de cualquier capa

$$\mathbb{E}\left[t\right] = \sum_{\mathbf{t}, e \in \{\mathbf{T}, \mathbf{E}\}} \delta_{\mathbf{t},t} e \tag{5.221}$$

donde

$$\delta_{\mathbf{t},t} = \begin{cases} 1\,, & \text{si } \mathbf{t}{=}(\mathbf{s}^{src}, [t_p, t_{p-1}, \ldots, t_1]) \wedge t \in [t_p, t_{p-1}, \ldots, t_1] \\ 0\,, & \text{en caso contrario} \end{cases} \tag{5.222}$$

indica si la transición t forma parte de la hiper-transición $\mathbf{t}$. De aquí se deduce que la llamada al método

$$\Lambda.\texttt{addExpectation}\left(\mathbf{t}, e\right)$$

[27] Para evitar problemas de estimación, los histogramas utilizados para la estimación de las distribuciones de cada modelo son inicializados con cuentas unidad.

donde $\mathbf{t} = (\mathbf{s}^{src}, [t_p, t_{p-1}, \ldots, t_1])$ es una hiper-transición y e su esperanza, conllevará para cada uno de los modelos

$$\lambda_i = \lambda\left(src\left(t_i\right)\right) = \lambda\left(dst\left(t_i\right)\right) \ , \quad t_i \in [t_p, t_{p-1}, \ldots, t_1] \tag{5.223}$$

la llamada al método

$$\lambda_i.\texttt{addExpectation}\left(t_i, e\right)$$

Por otro lado, y dado que toda hiper-transición de nivel p es debida a que los estados de las capas inferiores (1 a $p-1$) del hiper-estado $\mathbf{s}^{src} = [s_1^{src}, \ldots, s_N^{src}]$ eran finales, la llamada al método

$$\Lambda.\texttt{addExpectation}\left(\mathbf{t}, e\right)$$

conllevará para cada uno de los modelos

$$\lambda_i = \lambda\left(s_i^{src}\right) \ , \quad s_i^{src} \in \left[s_1^{src}, \ldots, s_{p-1}^{src}\right] \tag{5.224}$$

la llamada al método

$$\lambda_i.\texttt{addFinalExpectation}\left(s_i^{src}, e\right)$$

Por último, y siguiendo un razonamiento análogo a los anteriores, la llamada al método

$$\Lambda.\texttt{addFinalExpectation}\left(\mathbf{s}, e\right)$$

donde $\mathbf{s} = [s_1, \ldots, s_N]$ es un hiper-estado y e su esperanza final, conllevará para cada uno de los modelos

$$\lambda_i = \lambda\left(s_i\right) \ , \quad s_i \in [s_1, \ldots, s_N] \tag{5.225}$$

la llamada al método

$$\lambda_i.\texttt{addFinalExpectation}\left(s_i, e\right)$$

3. **Volcado de estadísticos**. Cuando al modelo se le notifica que el proceso de estimación de las transiciones ha finalizado, este debe aplicar las fórmulas de re-estimación de parámetros. Por ello, la llamada al método

$$\Lambda.\texttt{dumpSuffStats}\left(\right)$$

conllevará la llamada al método

$$\lambda.\texttt{dumpSuffStats}\left(\right)$$

para todos los modelos de todas las capas.

5.5. Decoder: Decodificación acoplada

La combinación de un sistema unificado de decodificación (Sección 5.1) con el formalismo de los modelos por capas (Sección 5.4) permite implementar una extensa (podríamos decir que infinita) familia de sistemas de decodificación. El Procesador Decoder implementa la decodificación en base a cualquier modelo que implemente la interfaz WFSA, y por lo tanto, también en base a un LMM. Para crear dichos modelos haremos uso del Comando LMM, que toma una serie de conjuntos de modelos y crea con ellos un LMM a medida. La capa del extremo superior puede formarse con un modelo que se suministre en línea de comando o puede generarse también de manera automática. La generación automática contempla dos casos básicos en los que no se precisa de información adicional alguna: el modelo *selector*, que permite seleccionar una única unidad de la capa inmediatamente inferior[28]; y el modelo *uniforme*, que establece que toda secuencia de modelos inferiores es equiprobable[29].

Además de los sistemas ya comentados en la sección anterior, podríamos diseñar cientos de sistemas para otras tantas tareas. Pongamos dos ejemplos más:

- **Categorizador de noticias**. La práctica totalidad de los sistemas anteriormente comentados se basaban en el procesamiento del habla. Sin embargo, el Procesador Decoder es compatible tanto con observaciones numéricas como textuales, y existen diversas herramientas de modelado de secuencias de texto, tales como sencillos WFSA deterministas, HMMs con distribuciones discretas o modelos KTLSS (modelos de n-gramas). Un categorizador de noticias podría construirse a partir de una capa de modelos de lenguaje (n-gramas, por ejemplo), sobre la cual podría colocarse una capa selectora.

- **Segmentador de audio multilingüe**. Supongamos que se tienen segmentos de audio con habla en diferentes lenguas y que se quiere hacer una segmentación según la lengua. Partiendo de conjuntos de modelos acústicos de cada lengua objetivo, sería posible construir un modelo de 3 capas. En la primera capa colocaríamos los modelos acústicos y en la segunda podríamos colocar un modelo de distribuciones de secuencias fonéticas por lengua. Podrían ser simples modelos de histogramas, o modelos de n-gramas que añadirían información fonotáctica al decodificador. Finalmente, y como el punto de partida era la segmentación, debemos suponer que como salida deberán producirse secuencias de segmentos en distintas lenguas, por lo que se debería añadir una tercera capa que determine de qué manera pueden generarse dichas secuencias. Si no se dispone de más información, podría utilizarse una distribución uniforme no normalizada.

Uno de los aspectos interesantes del mecanismo de decodificación mediante LMMs es que prácticamente toda la información con la que cuenta el sistema de decodificación

[28] El modelo es implementado mediante un autómata de dos estados y transiciones uniformes no normalizadas ($p(t) = 1$) del primer estado al segundo, cuya probabilidad final es 1. En la distribución resultante, toda secuencia de longitud 1 tendrá probabilidad 1, y el resto tendrá probabilidad 0.

[29] El modelo es implementado mediante un autómata de un único estado y transiciones uniformes no normalizadas ($p(t) = 1$) sobre sí mismo. Su probabilidad final es también 1. En la distribución resultante, toda secuencia tendrá probabilidad 1.

reside en los modelos que el usuario debe describir formalmente, evitando así que información que pudiéramos considerar "de modelado" pase a formar parte del algoritmo de decodificación[30]. Un caso ilustrativo es el de un decodificador acústico-fonético (DAF), donde los desarrolladores comúnmente tienden a tomar una decisión sobre la distribución previa de las secuencias de fonemas, decisión que suele ser incrustada directamente (*hard-coded*) en el código del decodificador. En cambio, si tratamos de generar un DAF en el entorno Sautrela, nos veremos forzados a formalizar explícitamente cuál es la distribución previa de las secuencias de fonemas. Lo que en un principio podría parecer una tarea tediosa y automática, se convierte a la postre en una más que interesante libertad de diseño que permite adaptar el proceso de decodificación a las necesidades concretas de cada situación.

La única restricción para formar una capa de conocimiento es que todos los modelos compartan un mismo alfabeto en común, lo que hace también posible integrar modelos de diferente naturaleza en una misma capa. Tampoco existe límite alguno en el número de capas que contiene un LMM, aunque lógicamente la complejidad computacional aumentará a medida que se añadan capas.

5.5.1. Transformación afín del logaritmo de las probabilidades

Si tomamos un sistema de dictado estándar, normalmente suele contar con un conjunto de parámetros que regulan las probabilidades de cada capa de conocimiento, así como ciertas penalizaciones de inserción. Dichos parámetros tratan de adaptar capa a capa el rango dinámico del logaritmo de la probabilidades de los modelos, y puede formalizarse de manera global como una transformación afín

$$\alpha_i \cdot \log p\left(x|\phi_i \in \Phi_i\right) + \beta_i$$

aplicada al logaritmo de las probabilidades. El término lineal α_i regula la información proveniente de cada capa, otorgando mayor relevancia a aquella capa que tenga mayor coeficiente α_i. El término constante suele ser interpretado como una penalización de inserción, ya que a menudo toma valores negativos, lo que confiere la capacidad de penalizar el número de veces que es evaluada la probabilidad[31]. Los LMMs tienen la capacidad de establecer una transformación afín para cada una de las capas. El Comando LMM permite establecer dichos parámetros en el momento de crear el LMM.

5.6. Trainer: Entrenamiento acoplado

De manera análoga a lo dispuesto en la sección anterior, el mecanismo unificado de entrenamiento de WFSAs (Sección 5.2) es complementado por la capacidad de los LMMs de implementar la interfaz de entrenamiento (Sección 5.4). De poco serviría la interfaz de entrenamiento (que se limita a un único modelo) sin los LMMs, y

[30] Aunque, como se analiza en la Sección 5.7, esta separación también puede conllevar ciertos problemas.

[31] Para la primera capa, y dado que su emisión debe ser evaluada para cada muestra de entrada, el término constante no modifica el proceso de decodificación. Para un LMM con una capa inicial acústica, por ejemplo, no tiene sentido establecer una penalización acústica, ya que las observaciones acústicas vienen dadas.

difícilmente sería posible plantear el entrenamiento de un LMM sin esta interfaz de entrenamiento. Como ya se ha mostrado en la Sección 5.4.6 que la implementación de la interfaz de entrenamiento de los LMMs es extremadamente sencilla.

5.6.1. Mecanismo de entrenamiento

En la fase de entrenamiento, las capas superiores de un LMM permiten formalizar el tipo de supervisión que gobernará la estimación de las capas inferiores. Sin embargo, el entrenamiento de modelos secuenciales requiere normalmente llevar a cabo el entrenamiento frente a conjuntos de secuencias con supervisión no uniforme. El calificativo "no uniforme" se refiere a que las secuencias contarán posiblemente con cierta información de supervisión que variará de una secuencia a otra. En el caso de entrenamiento de modelos acústicos, por ejemplo, lo normal es contar con un conjunto de segmentos de audio que corresponden a diversas locuciones de un conjunto de locutores, donde a cada locutor se le ha solicitado que lea un conjunto de textos. En otras palabras, cada secuencia de datos contará con una transcripción ortográfica distinta. Ello implica que los LMMs a entrenar serán distintos. No obstante, dicha situación está contemplada en el entrenador de Sautrela.

El Procesador Trainer permite realizar la re-estimación de un conjunto de N capas de conocimiento, donde cada capa queda representada por un conjunto de WFSAs. El procesador añadirá, para cada secuencia de entrada, una capa de supervisión obtenida a partir de la cabecera de datos de la secuencia. Los pasos a seguir son:

1. A partir del valor de la propiedad `transcPropertyName` del procesador, extrae de la cabecera de datos una cadena de caracteres.

2. Utiliza el valor de la propiedad `transcPropertySplit` del procesador como expresión regular para trocear la cadena de caracteres. La secuencia de cadenas resultante representa una secuencia de clases a entrenar, donde cada clase corresponde con un modelo de la capa superior suministrada.

3. Con la transcripción obtenida se crea un sencillo autómata determinista que podrá tener inserciones de un símbolo adicional (propiedad `transcInsertionSymbol`). Dicha inserción es configurable al inicio, al final o entre los otros símbolos (propiedad `transcInsertionMask`).

4. Añade la capa de supervisión, produciendo un LMM de $N + 1$ capas.

De esta manera, para cada secuencia de datos es utilizado un LMM diferente, pero con un subconjunto común de N capas. A los modelos de dichas capas les serán transmitidas las esperanzas de las transiciones correspondientes y una vez procesado el conjunto completo de secuencias de entrada, finalizará la fase de re-estimación.

A continuación, retomaremos un ejemplo planteado en la Sección 5.4: el entrenamiento de modelos acústicos a partir de transcripciones ortográficas. Supongamos que contamos con la transcripción ortográfica de las frases locutadas o, para ser más precisos, la representación ortográfica mostrada a los locutores, ya que nadie nos asegura que realmente hayan pronunciado esa frase. Mediante una herramienta de traducción

grafema-fonema (un transcriptor fonético) es posible obtener una transcripción canónica de cada palabra, obteniendo una secuencia fonética para cada secuencia de palabras (secuencia léxica). Si integramos ambas informaciones (las secuencias léxica y fonética) en la base de datos acústica (véase Sección 4.4) tenemos ya todo dispuesto para comenzar el entrenamiento. Analizaremos varias maneras de abordar la re-estimación, que se basarán en las dos fuentes de supervisión que hemos añadido:

- **Secuencia fonética**. Es normalmente la mejor elección si partimos de modelos sin información acústica previa. Es de suponer que la transcripción fonética no representa demasiado bien la secuencia acústica. Sin ir más lejos, no cuenta con silencios. Se pueden hacer diversas suposiciones sobre su ubicación: al inicio y final de la frase, entre la palabras, entre los fonemas, etc. Las propiedades `transcInsertionSymbol` y `transcInsertionMask` están justamente pensadas para dar cabida a eventos no transcritos (como los silencios). El modelo de supervisión que crea el Procesador Trainer permite la inserción opcional del símbolo `transcInsertionSymbol` en base al criterio de inserción `transcInsertionMask`, que puede indicar la inserción inicial (`l` de left), interna (`i` de internal) o final (`r` de right). Podemos proponer dos configuraciones:

  - Silencios opcionales al inicio y final de frase (`lr`). En una fase inicial de entrenamiento es conveniente limitar al máximo la flexibilidad de la capa de supervisión. Una vez estimados unos modelos iniciales, se permitirá una mayor libertad.
  - Silencios opcionales al inicio, final y entre todos los fonemas de la frase (`lir`). Téngase en cuenta que la inserción es opcional, no forzosa. Si previamente se ha aplicado la configuración anterior, es de esperar que sean justamente los modelos de silencio los que estén mejor estimados, por lo que parece sensato suponer que su libre inserción ayude a una mejor estimación del resto de modelos.

- **Secuencia léxica**. Supongamos ahora que contamos con un diccionario de transcripciones o, más genéricamente, un conjunto de modelos léxicos del conjunto de unidades léxicas presentes en las frases. Al añadir dicha información podríamos obtener una mejor estimación de los modelos acústicos, ya que no estaríamos imponiendo la transcripción canónica de cada palabra. A diferencia del caso anterior, ahora re-estimaríamos un modelo que consta de tres capas: la acústica, la léxica y la capa de supervisión que sería generada automáticamente. Una vez más, nos encontraremos con el inconveniente de los silencios, que en este caso podrían ser considerados una unidad léxica[32] (permitiendo la inserción libre de silencios entre palabras, `lir`) o una unidad acústica (añadiendo el silencio opcional al final de cada modelo léxico[33]).

[32] Coexistirían el silencio como unidad léxica y el silencio como unidad acústica, pero su inserción sería definida a nivel léxico.

[33] Aun así, seguiría siendo necesario la definición del silencio como unidad léxica, ya que el formalismo anterior no daría cobertura al silencio inicial de frase. En tal caso, solo habría que permitir la inserción de silencios al inicio de la frase (opción `l`).

Un aspecto interesante a considerar para cualquiera de las capas de supervisión analizadas es que si, una vez hayan convergido los modelos acústicos, utilizamos esa misma capa para decodificar cada frase, obtendríamos una transcripción más fiel de lo que realmente ha sido pronunciado, e incluso una posible segmentación fonética. Cualquiera de las dos informaciones puede ser utilizada como punto de partida para la estimación de un conjunto de modelos alternativo y, posiblemente, más complejo.

5.6.2. Entrenamiento simultáneo de múltiples capas

En el apartado anterior, al analizar el entrenamiento de modelos acústicos con supervisión basada en modelos léxicos, se ha pasado por alto el hecho de que suministramos dos capas de conocimiento al entrenador, y que este re-estimará ambas capas, no solo la capa acústica. Es decir, estaríamos re-estimando simultáneamente los modelos léxicos y los modelos acústicos. En el caso concreto anterior, sin embargo, no estaríamos interesados en la re-estimación de los modelos léxicos, ya que posiblemente ni siquiera representen el vocabulario que después (en la aplicación) tendremos que utilizar con los modelos acústicos entrenados. Por ello, lo lógico sería descartar dichas re-estimaciones.

Pero cabe considerar que dichas estimaciones existen[34], y pueden ser una interesante fuente de información en otro tipo de sistema. Supongamos, por ejemplo, que deseamos implementar un segmentador de audio multilingüe como el descrito en la Sección 5.5. Estará compuesto por:

1. Una capa acústica con un conjunto de unidades acústicas que dé cobertura al conjunto de lenguas objetivo.
2. Un modelo fonotáctico (n-gramas de fonemas, por ejemplo) por lengua.
3. Un modelo uniforme sobre el conjunto de secuencias de lenguas.

Podríamos haberlos entrenado los modelos acústicos a partir de un conjunto de bases de datos acústicas de cada lengua, y los modelos fonotácticos podrían haberse estimado a partir de texto transcrito a secuencias fonéticas. El modelo uniforme no tiene parámetros, por lo que no requiere estimación. Pues bien, todavía tenemos la oportunidad de realizar una re-estimación acoplada de los modelos. Para cada una de las lenguas podemos definir un LMM de dos capas compuesto por el conjunto de unidades acústicas de dicha lengua[35] y el correspondiente modelo fonotáctico, y re-estimarlo frente a la base de datos acústica de dicha lengua. Una vez finalizadas todas las re-estimaciones, podemos crear el LMM de tres capas anterior o incluso un LMM de dos capas compuesto por un conjunto de LMMs y un modelo uniforme (ambas configuraciones son

[34] Para ser más precisos, la realidad es que incluso la capa de supervisión es re-estimada, aunque en este caso es el propio Procesador Trainer quien descarta dicha información, ya que la capa de supervisión es creada (y destruida) dinámicamente.

[35] Por simplificar el análisis, supondremos que los conjuntos de unidades acústicas son disjuntos. Esta es una cuestión que solo debe ser entendida como una decisión de modelado, no lingüística, y que, de hecho, ocurre de manera natural, ya que seguramente contaremos con un conjunto de unidades acústicas entrenado para cada lengua objetivo. Ello no provoca ningún conflicto, ya que el nombre de las unidades es arbitrario, y pueden coexistir por ejemplo `"a_ES"` y `"a_EUS"`, en referencia a los fonemas "`a`" del castellano y del euskera, respectivamente.

equivalentes, si bien la primera resulta computacionalmente más "ligera"). Analicemos la diferencia entre ambos métodos de estimación de modelos:

- **Método desacoplado**: Los modelos acústicos $p(X|\phi_f)$ han sido entrenados para maximizar la verosimilitud del conjunto de secuencias de observaciones $X = (x_1, \ldots, x_N)$, pero condicionados a cierta información de supervisión o transcripción que podríamos considerar "ruidosa" y que establecía cuál era la secuencia latente de fonema $F = (f_1, \ldots, f_m)$ de cada secuencia de observaciones. El modelo fonotáctico $p(F|\psi)$ ha sido entrenado para maximizar ciertas secuencias de fonemas $F = (f_1, \ldots, f_k)$ obtenidas a partir de texto (no habla) que ha sido transcrito fonéticamente de manera automática. Poco podemos decir sobre la distribución marginal

$$p(X|\psi) = \sum_{\forall F} \left(\prod_{f_i \in F} p(X|\phi_{f_i}) \right) \cdot p(F|\psi) \tag{5.226}$$

  derivada de la composición de ambas capas, y que en definitiva es una medida de ajuste de los modelos a la tarea de segmentación.

- **Método acoplado**: La re-estimación para cada lengua del LMM compuesto por los modelos acústicos y el modelo fonotáctico busca justamente la optimización de la verosimilitud de la Ecuación 5.226, por lo que en este caso podemos asegurar, en base al algoritmo EM, que el conjunto de modelos re-estimado de manera acoplada obtendrá una mayor verosimilitud que el anterior. Las estimaciones desacopladas previas son un buen punto de partida para la inicialización de los parámetros; sin embargo, sus estimaciones pueden ser mejoradas.

Un factor a tener en cuenta al plantear re-estimaciones acopladas es que por la mera naturaleza de los modelos, el número de apariciones de los elementos de las capas decrece a medida que ascendemos de nivel. Dada una base de datos acústica que contenga una hora de audio, podríamos encontrar:

- 360000 vectores de muestras
- 60000 fonemas $\xrightarrow{|fon|=60}$ 10000 muestras por fonema
- 10000 palabras $\xrightarrow{|pal|=10000}$ 1 muestra por palabra

Esto indica que una re-estimación acoplada de un conjunto de modelos acústicos y léxicos es a priori desaconsejable. No obstante, debemos recordar que existe la posibilidad de plantear una re-estimación MAP de los conjuntos de modelos (véase Sección 5.3), donde solo se verían afectados los parámetros asociados a las transiciones que acumulen suficiente esperanza posterior. Nótese que la frecuencia de aparición aportada es un promedio: habrá multitud de palabras que no aparezcan y un pequeño conjunto de unas 100 palabras que acaparen gran parte de la esperanza. Dado que esas mismas palabras volverán a aparecer más frecuentemente durante la decodificación futura, resulta más que interesante que sus modelos léxicos puedan ser re-estimados[36].

[36] Suponemos aquí que la base de datos cubre adecuadamente la variabilidad léxica que esperamos

5.7. TreeModel: Búsqueda basada en árbol léxico

Sin lugar a dudas, la decodificación y re-estimación unificada de modelos a través de interfaces es una de las principales aportaciones de Sautrela. La capacidad de interaccionar con un modelo y estimar la esperanza de sus transiciones sin necesidad de conocer su naturaleza interna es la clave de ambos mecanismos. Sin embargo, el desconocimiento del gobierno interno de un modelo conlleva ciertos retos a la hora de implementar ciertos mecanismos algorítmicos que incluyen los entornos de desarrollo tradicionales.

Uno de los más notables lo encontramos en los sistemas de dictado de gran vocabulario. Al considerar el final de una palabra, el decodificador debe considerar todas las posibles palabras sucesoras, lo que genera una explosión de posibles alternativas a explorar. Cuando un decodificador se rige por este "espacio de búsqueda" se dice que implementa un léxico lineal. Si bien es cierto que una búsqueda en haz reducirá el espacio de búsqueda, debemos tener en cuenta que en principio no deberíamos aplicar la poda hasta haber evaluado la probabilidad acústica correspondiente, ya que podría darse el caso de que la información previa del modelo de lenguaje no fuera acorde a la evidencia acústica (una poda prematura basada únicamente en las probabilidades del lenguaje puede provocar una distorsión severa en el espacio de búsqueda).

Cuando se manejan léxicos de gran tamaño, una alternativa al modelado independiente de cada palabra (cada una cuenta con su correspondiente modelo de secuencias subléxicas) es la representación en árbol que se muestra en la Figura 5.9. En cierta manera, esta estructura resulta natural a la hora de representar un léxico de grandes dimensiones, ya que permite "compartir" una única estructura de modelado entre todos los modelos léxicos. Cada nodo o estado del árbol léxico representa el hecho de encontrarse simultáneamente en diversos modelos léxicos. No obstante, es preciso hacer algunas observaciones:

- No es posible representar cualquier conjunto de modelos léxicos mediante un árbol léxico. El árbol léxico de la Figura 5.9 representa a un conjunto de modelos deterministas sin bucles.

- En un decodificador basado en una representación de léxico lineal, la probabilidad de los modelos léxicos es aplicada al transitar de un fonema a otro. Sin embargo, al recorrer un camino del árbol léxico estamos considerando simultáneamente el conjunto de palabras con un prefijo común. Debido a este hecho, no es posible transferir directamente las probabilidades de los modelos léxicos a las transiciones fonéticas. Sin embargo, sí que es posible tomar las probabilidades que los modelos léxicos asignan a las secuencias de fonemas y realizar una propagación hacia atrás similar a la estimación de coeficientes de retroceso β de la Sección 5.2.5. La normalización de la aportación de los coeficientes β a cada nodo da lugar a una distribución de probabilidad que puede ser utilizada en la decodificación: dado un camino a través del árbol, el producto de dichas probabilidades será igual

encontrar posteriormente, ya que de lo contrario estaríamos sobre-entrenando los modelos, esto es, ajustándonos demasiado al dominio de entrenamiento. Por ejemplo, no tendría ningún sentido re-estimar modelos léxicos con una variedad dialectal de una lengua y pretender reconocer otra variedad cuyas características léxicas difieren claramente.

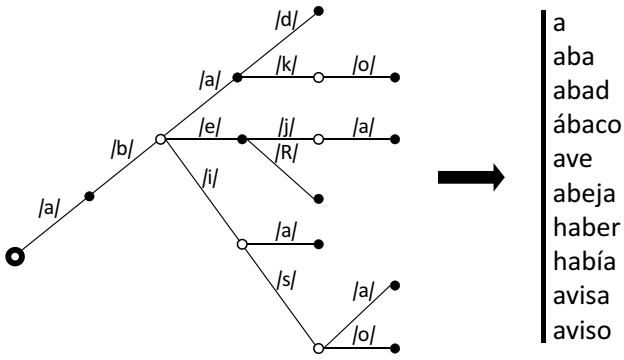


Figura 5.9 Ejemplo de representación en árbol de un léxico compuesto por 10 palabras, donde cada rama representa una unidad fonética y los nodos sombreados representan finales de palabra.

a la probabilidad que el correspondiente modelo léxico asigna a la secuencia de fonemas.

- Análogamente a lo explicado para los modelos léxicos, no es posible aplicar la probabilidad del modelo de lenguaje al inicio del modelo léxico. Existe, sin embargo, la posibilidad de crear tantos árboles léxicos como estados tenga el modelo de lenguaje, en cuyo caso sería posible estimar de antemano una propagación conjunta de las probabilidades de los modelos léxicos y la probabilidad del modelo de lenguaje. La Figura 5.10 muestra un ejemplo ilustrativo de la propagación conjunta de ambas distribuciones de probabilidad. Dado un camino que termine en un nodo final de una palabra, el producto de todas las transiciones es igual al producto de las probabilidades de ambos modelos. [37]

La búsqueda basada en árbol léxico puede reducir casi en un orden de magnitud el coste temporal del proceso de decodificación [113, 73]. Sin embargo, como se ha podido comprobar, este método requiere del conocimiento expreso de la naturaleza de los modelos que conforman el decodificador. Recordemos que la decodificación unificada de Sautrela no considera la existencia de los modelos léxicos o de lenguaje, ya que realiza la decodificación frente a un único WFSA (es el LMM quien se encarga de formalizar la composición de capas como un único autómata). Parece, por tanto, imposible formalizar un método unificado de decodificación capaz de desarrollar una búsqueda basada en árbol léxico.

No obstante, y como ocurre con los LMMs, la solución viene de la mano de la formalización de un modelo cuya decodificación unificada sea equivalente a la búsqueda

[37] También es posible reducir el número de árboles léxicos pre-estimados. Si el modelo de lenguaje se basa en 3-gramas, por ejemplo, el número de estados será igual al número de 2-gramas considerados, pero podría limitarse el número de árboles al número de unigramas, en cuyo caso el árbol representaría una aproximación de la probabilidad del modelo de lenguaje y al final de cada palabra existiría una probabilidad de ajuste que compense la diferencia entre el modelo de 2-gramas y el modelo de 3-gramas. De manera similar, se puede construir un único árbol léxico que contenga solo las probabilidades de 1-gramas (la frecuencia de cada palabra).

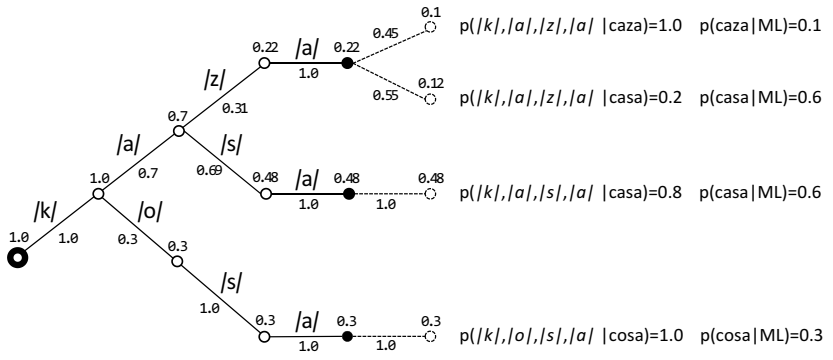


Figura 5.10 Ejemplo de un árbol léxico con probabilidades de transición que representa el vocabulario $\{caza, casa, cosa\}$, donde la secuencia de fonemas $(|k|, |a|, |z|, |a|)$ puede corresponder a las palabras *caza* y *casa*. Las transiciones adicionales (líneas discontinuas) que parten de los nodos sombreados representan transiciones finales a cada una de las palabras que finalizan en dicho nodo. Para cada final de palabra se indica tanto la probabilidad $p(\cdot|pal)$ del modelo léxico como la probabilidad $p(\cdot|ML)$ del modelo de lenguaje. Los coeficientes de retroceso están situados encima de cada nodo y sus valores iniciales corresponden al producto de las probabilidades léxica y de lenguaje. Las probabilidades de transición están situadas debajo de cada transición. El coeficiente de retroceso de un nodo es la suma de los coeficientes que llegan hasta él, y la probabilidad se obtiene mediante la normalización de cada coeficiente por la suma de todos ellos. El producto de las probabilidades de transición de cualquier camino a través del árbol es igual al producto de las probabilidades del modelo léxico y del modelo de lenguaje.

en árbol léxico. Sautrela define el modelo TreeModel, el cual integra en un único autómata no determinista dos capas de conocimiento compuestas de un diccionario en modo de árbol léxico y un modelo de lenguaje. De manera análoga al LMM, un TreeModel define hiper-estados que representan simultáneamente el estado del modelo de lenguaje y el estado del árbol léxico. La implementación de Sautrela hace uso de un único árbol con propagación de probabilidades léxicas[38] y probabilidades de lenguaje incontextuales[39].

Un modelo TreeModel puede ser utilizado en lugar de las dos capas que antes representaban el modelo de lenguaje y el modelo léxico. Así, un sistema de dictado podría constar de una capa inferior con el conjunto de modelos acústicos y una capa superior con un TreeModel que implementa la búsqueda en árbol. Además, como el TreeModel es un WFSA más, es posible integrar un conjunto de ellos en una única capa. Por ejemplo podríamos crear un reconocedor multilingüe que suponga que cada secuencia de muestras de entrada pertenece a una única lengua. Dicho reconocedor

[38] La implementación actual no incluye probabilidades en el diccionario, por lo que la probabilidad léxica es establecida como $\frac{1}{|L_i|}$, conde $|L_i|$ representa el número de transcripciones alternativas de cada palabra.

[39] El modelo de lenguaje es consultado sobre la probabilidad de observar cada símbolo del alfabeto al inicio de una secuencia, lo que, en función de la implementación, es equivalente a la probabilidad de unigrama en un modelo de n-gramas.

estaría compuesto por una capa con el inventario completo de modelos acústicos, otra capa con varios TreeModels, uno por lengua, y una capa superior selectora.

En el Apéndice B puede encontrarse más información sobre el Comando TreeModel que permite la creación de modelos con representación de léxico en árbol.

Capítulo 6

Contribuciones

Los capítulos anteriores se han centrado en mostrar la estructura de Sautrela, con las decisiones de diseño y las aportaciones que incorpora a las TH, explicando tanto sus bases teóricas como los aspectos propios de su implementación. El entorno, que ha sido desarrollado íntegramente por el aspirante, ha sido tanto un objetivo en sí mismo como un elemento de utilidad en toda su trayectoria investigadora, que abarca numerosos trabajos. El presente capítulo enumera y describe sucintamente el conjunto de contribuciones en las que ha tomado parte el aspirante a lo largo de su carrera investigadora.

6.1. Reconocimiento automático del habla

El inicio en la investigación del aspirante tuvo que ver con el desarrollo de motores de reconocimiento automático del habla (RAH) en lengua española [166, 165]. Los sistemas desarrollados contaban con una arquitectura cliente-servidor, de manera que en la parte del cliente se contaba con una interfaz gráfica y se realizaba la parametrización de la señal, mientras que la parte del servidor se ocupaba de la búsqueda de la secuencia más probable de palabras dada una secuencia de vectores de parámetros.

Poco después, en un momento en que se planteaba la idoneidad de diferentes conjuntos de unidades fonéticas (semifonemas, trifonemas, etc.), comenzó una nueva línea de investigación basada en el uso de unidades léxicas alternativas a la palabra en sistemas de RAH [120, 127]. Las unidades léxicas alternativas ofrecen cierto grado de libertad a la hora de definir el conjunto de unidades léxicas a reconocer, el cual puede ser optimizado en base a criterios que combinen la perplejidad del modelo de lenguaje y el tamaño del conjunto de unidades. En [127] se partía del conjunto de unidades acústicas (fonemas), el cual era ampliado mediante la agrupación sucesiva de unidades, en base a diferentes heurísticos. Los primeros resultados demostraron que era posible reducir la tasa de error a nivel de frase. No obstante, en la mayoría de los casos las palabras son un fin en sí mismo, bien porque se trate de un sistema de dictado, bien porque la salida del reconocedor es postprocesada por algún sistema que espera una entrada en forma de secuencias de palabras. Para comparar el rendimiento de las unidades léxicas alternativas en este contexto, se planteó un sencillo alineamiento a palabras

de la secuencia fonética derivada de la secuencia de unidades léxicas reconocida. Pudo comprobarse que mediante esta sencilla conversión de unidades léxicas a palabras, la tasa de error a nivel de palabra del sistema resultante era menor que la del sistema de referencia. Motivado por este hecho, en [120] se analizaron otros métodos de conversión de la secuencia de unidades léxicas a una secuencia de palabras, mejorando el rendimiento en términos de la tasa de error a nivel de palabra.

Posteriormente el uso de unidades léxicas fue aplicado a un sistema de RAH en euskera [106, 105]. Al tratarse de una lengua aglutinante, las unidades alternativas a la palabra reducían drásticamente el vocabulario de la tarea, permitiendo reducir el coste computacional de la búsqueda de la secuencia más probable de unidades. Las unidades léxicas alternativas fueron tenidas en cuenta también en [33, 34, 30, 31, 32] a la hora de crear un corpus para desarrollar sistemas de RAH orientados a la indexación de recursos de noticias bilingües en español y euskera.

Más recientemente, se ha colaborado en la elaboración y publicación de una base de datos en euskera destinada a las tecnologías del habla, que se compone de dos subconjuntos de datos: por un lado, un corpus grabado en entorno de oficina y, por otro lado, un corpus grabado a través de canal telefónico [114].

6.2. Sistemas Tutores Inteligentes

En [107, 70, 172, 50, 87, 88] se colaboró en el desarrollo de sistemas tutores inteligentes (STI) para la integración laboral de personas con discapacidades, centrándose sobre todo en personas con síndrome de Down. En los desarrollos realizados, el STI fue concebido como un asistente integrado en una plataforma móvil (PDA o teléfono móvil), configurable por el tutor humano y que debía acompañar el proceso de aprendizaje de una tarea laboral por parte de la persona con discapacidad. El desarrollo incorporaba un sencillo sistema de diálogo para facilitar la interacción con el usuario.

6.3. Verificación de la lengua

Desde 2007, el grupo Grupo de Trabajo en Tecnologías Software (GTTS) ha participado en las evaluaciones internacionales de verificación de la lengua organizadas por el Instituto Nacional de Estándares y Tecnologís (NIST, por sus siglas en inglés) del Departamento de Comercio de los EEUU. La primera participación fue meramente testimonial [125], pero abrió el camino de una ininterrumpida actividad en esta tarea [141] que culminó con unos excelentes resultados en la última edición [139, 140, 162, 163].

La experiencia obtenida en las competiciones del NIST animó al aspirante y al resto del grupo de investigación a organizar evaluaciones nacionales de verificación de la lengua en el contexto de las Jornadas de Tecnologías del Habla. Dichas evaluaciones, denominadas Albayzin Language Recognition Evaluation, han contado con tres ediciones, y han dado lugar también a la creación de diferentes corpus de evaluación de las tecnologías de verificación de la lengua:

- En 2008 se organizó la primera de las evaluaciones, Albayzin-LRE 2008, que centro su objetivo en la verificación de las cuatro lenguas oficiales españolas:

Español, Catalán, Euskera y Gallego [154]. Para llevar a cabo esta evaluación se creo el corpus KALAKA [155, 185] que contaba con alrededor de 8 horas de entrenamiento para cada lengua objetivo, casi 8 horas de desarrollo y otras tantas de evaluación (estas últimas en segmentos de duración nominal de 30, 10 y 3 segundos), extraídas de 30 horas de grabación de programas de televisión para cada una de la lenguas objetivo, y unas 18 horas adicionales de otras lenguas (Alemán, Francés, Inglés y Portugués).

- En 2010 se organizó la segunda edición, Albayzin-LRE 2010, tomando como base la anterior evaluación, pero ampliando el conjunto de lenguas al Inglés y Portugués, y añadiendo condiciones de entorno ruidoso [158, 156]. Para este evento se creó el corpus KALAKA-2 [159], con un mayor volumen de datos e incluyendo segmentos ruidosos. Además, y dado que el Inglés y el Portugués pasaban a ser lenguas objetivo, el conjunto de lenguas externas fue también ampliado con el Árabe y el Rumano.

- En 2012 se organizó la última edición, Albayzin-LRE 2012. En esta ocasión, la evaluación se enfocó a la verificación de la lengua de vídeos contenidos en la plataforma YouTube, diferenciando además dos condiciones de desarrollo: en la primera de ellas, *Plenty*, el participante contaría con un gran corpus de entrenamiento de las lenguas objetivo, pero en la segunda, *Empty*, los materiales referidos a las lenguas objetivo se verían reducidos a un pequeño conjunto de desarrollo [151, 167, 150]. Para llevar a cabo esta evaluación se creó el corpus KALAKA-3 [160, 161] que se diferenciaba de las anteriores versiones en dos aspectos fundamentales:

  - Los segmentos de audio de evaluación correspondían a vídeos completos sin editar, por lo que no tenían una duración nominal, sino que estaban acotados a una duración mínima y máxima de 30 y 120 segundos, respectivamente (con una duración nominal mínima de 5 segundos).
  - Las lenguas objetivo de la última edición (Español, Catalán, Euskera, Gallego, Inglés y Portugués) fueron utilizadas para la condición Plenty, mientras que la condición Empty contó con un nuevo conjunto de lenguas objetivo: Francés, Alemán, Griego e Italiano. El conjunto de lenguas externas estaba compuesto por otras 11 lenguas europeas.

Las evaluaciones Albayzin-LRE han dado lugar también a la definición de nuevas métricas de evaluación de sistemas de verificación de la lengua [150] que a diferencia de las métricas utilizadas por el NIST, no representan un coste basado en los errores de decisión del sistema, sino que evalúan de manera eficaz la cantidad de información aportada por este.

En lo que se refiere al desarrollo de tecnologías de modelado para la verificación de la lengua, en [135, 134, 133] se comenzó con una línea de investigación que trataba de fusionar de manera síncrona la información proveniente de distintos decodificadores fonéticos en sistemas fonotácticos para formar un único sistema fonotáctico multilingüe basado en co-ocurrencias de fonemas. La investigación partió de la consideración de la secuencia de fonemas más probable de cada decodificador (*1-best*) [138], sin embargo

en [181] se mostró cómo generalizar el método de las co-ocurrencias al caso de salidas con mallas (*lattices*) de fonemas. Posteriormente pudo comprobarse también que la información aportada por las co-ocurrencias podía ser aprovechada para mejorar el rendimiento de un sistema de verificación de dialectos [184].

Una de las limitaciones de los sistemas fonotácticos es que resulta muy difícil utilizar un conjunto de n-gramas fonéticos que pueda llegar a contener palabras o unidades superiores, ya que al aumentar el orden de los n-gramas, la explosión combinatoria hace que el espacio de parámetros se vuelva intratable. En [137, 136] se presentó un sencillo mecanismo de selección de n-gramas de fonemas, cuyo vector de parámetros resultante podía además ser proyectado a un subespacio sin una degradación significativa de los resultados finales.

Otro aspecto interesante de los sistemas de verificación de la lengua es su calibración y fusión. En [132] se estudiaron diferentes métodos de calibración de sistemas de verificación y en [164] se mostró un estudio de la fusión de multitud de sistemas heterogéneos que participaron en la evaluación Albayzin-LRE 2010.

Más recientemente se ha podido comprobar que la información obtenida por un sistema acústico de verificación de la lengua puede depender de la longitud de ventana de análisis. Así, en [53] se demostró que sistemas basados en diferentes tamaños de la ventana de análisis ofrecen informaciones complementarias susceptibles de ser fusionadas posteriormente.

La línea de investigación de verificación de la lengua ha sido integrada en el desarrollo de sistemas de indexado y búsqueda de recursos audiovisuales que se detalla en la Sección 6.6. No obstante también ha dado lugar a proyectos de investigación específicos, tales como el proyecto GLOSA, presentado en [171].

6.4. Reconocimiento, seguimiento y verificación del locutor

Los primeros trabajos de investigación en el área del reconocimiento o verificación del locutor en los que tomó parte el aspirante se remontan a 2006, donde se estudió la selección y pesado de parámetros acústicos en base a algoritmos genéticos [228, 230, 229]. Posteriormente, el método de selección y pesado fue comparado frente a una transformación no supervisada basada en PCA y otra transformación supervisada basada en LDA, comprobándose que para la condición de habla limpia (condiciones de laboratorio) los algoritmos genéticos mejoraban los resultados obtenidos con las transformaciones, pero no así en el caso de conversaciones telefónicas [235, 234, 236].

A partir del primer contacto con las evaluaciones de verificación de la lengua del NIST, el grupo GTTS comenzó a participar también en las evaluaciones de verificación del locutor organizadas por este mismo organismo, tomando parte en las ediciones de 2008 [239], 2010 [130, 131] y 2011 [54]. Posteriormente, en 2013, tomó parte en la evaluación MOBIO, orientada a aplicaciones móviles [90]. Por último, tomó parte en la NIST 2014 Speaker Recognition i-Vector Challenge, posteriormente denominada NIST i-vector Machine Learning Challenge y que trataba de facilitar el acceso de investigadores del área de aprendizaje automático de patrones a la tarea de verificación del locutor [128]. La participación continuada en dichas competiciones ha sido clave

a la hora de desarrollar diversas líneas de trabajo relacionadas con las tecnologías de reconocimiento, seguimiento y verificación del locutor.

Así, en [238, 237] se presentó un método de modelado que se denominó modelado superficial (*shallow source modeling*) que complementaba al UBM en aquellas situaciones en las que las características acústicas de la señal a evaluar no quedaban cubiertas por el UBM. Pudo demostrarse que la combinación del UBM y un modelo superficial aumentaba la robustez ante señales acústicamente extrañas sin suponer degradación alguna ante señales normales.

En [231, 232, 233] se investigó el uso de las tecnologías propias de un sistema de verificación del locutor para crear un sistema de seguimiento de locutor que pudiera ser integrado en un entorno de Inteligencia Ambiental. A diferencia de los sistemas de verificación, cuyo cometido es emitir una decisión sobre si una señal completa de audio corresponde a un locutor que puede estar representado por un conjunto de señales, el sistema de seguimiento desarrollado tenía por objeto obtener la esperanza de la identidad del locutor pero a intervalos predefinidos de 1 segundo y con una latencia mínima. Este sistema fue integrado en una arquitectura orientada a servicios (SOA, por sus siglas en inglés) que ofrecía dos servicios diferentes: un servicio de seguimiento de locutor de más alto nivel y otro servicio de detección de locutores de más bajo nivel.

En [129] se presentó el sistema con el que GTTS se presentó a la evaluación NIST-SRE 2010. Este sistema se basaba en uno de los mejores sistemas de la anterior edición, y contaba con un sencillo pero eficaz mecanismo de compensación de canal y obtención de puntuaciones (*scores*) basado en estadísticos suficientes. Esta tecnología fue posteriormente utilizada para construir un sistema de diarización [55, 56] en el que la similitud de los segmentos a agrupar era obtenida como producto de la verificación de locutor de ambos segmentos.

6.5. Nuevos parámetros para la verificación de la lengua y del locutor

En [59] se presentó un nuevo conjunto de parámetros acústico-fonéticos que integra información fonética a nivel de muestra (*frame*). Estos parámetros, denominados Phone Log-Likelihood Ratios (PLLR), han demostrado contener información complementaria frente a sistemas basados tanto en parámetros acústicos como en información fonotáctica. Su validez como parámetros en los sistemas de verificación de la lengua ha sido contrastada frente a diferentes conjuntos de evaluación [59, 60].

En [57] se analizaron posibles reducciones de dimensionalidad de los PLLRs basándose tanto en criterios supervisados de agrupación de fonemas que parten del conocimiento lingüístico, como en criterios no supervisados. Los resultados obtenidos indicaban que el mejor método de reducción de dimensionalidad era el basado en el análisis de componentes principales (PCA), que además no requería conocimiento lingüístico alguno. Posteriormente, en [64] se investigó la posibilidad de aplicar, a partir de un conjunto ya reducido de parámetros, la técnica denominada *Shifted-Delta*, comúnmente utilizada en los sistemas acústicos y que consiste en incorporar coeficientes dinámicos calculados alrededor de la ventana de análisis. Pudo comprobarse que esta técnica mejoraba aún más los resultados del sistema.

Por último, en [63] se analizó el espacio de parámetros de los PLLRs, comprobando que el hiperplano en el que residían los parámetros mostraba una tendencia asintóticamente perpendicular a los ejes de los parámetros, lo que provocaba que las distribuciones de parámetros tuviesen una naturaleza acotada nada adecuada para su modelado mediante distribuciones Gaussianas. En dicho trabajo se propuso una sencilla proyección que eliminaba la naturaleza acotada de los parámetros. Partiendo de los parámetros proyectados, su posterior reducción de dimensionalidad mediante PCA y aplicando *Shifted-Delta*s, en [61] se mejoraron aún más los resultados de verificación.

A pesar de que los PLLRs fueron concebidos para tareas de verificación de la lengua, en [58, 62] pudo comprobarse que, a pesar de no resultar tan efectivos, estos parámetros también aportan cierta información complementaria a los sistemas acústicos de reconocimiento del locutor.

6.6. Sistema de indexación y búsqueda de contenidos multimedia

Otra línea de trabajo llevada a cabo en el grupo de investigación y en la que el aspirante ha colaborado activamente es el desarrollo de sistemas de indexación y búsqueda de contenido multimedia. El contenido multimedia en el cual se ha basado esta investigación ha estado compuesto principalmente por vídeos de informativos televisivos o sesiones parlamentarias. En este desarrollo han confluido prácticamente el conjunto completo de líneas de investigación del grupo GTTS, ya que un sistema de indexado de contenido multimedia requiere de la existencia de diversos módulos que permitan extraer de manera incremental la información contenida en los recursos. Aunque no es la única configuración posible, un sistema de indexado puede contener diversas fases tales como: una segmentación acústica que descarte los segmentos que no contienen habla, una identificación de la lengua hablada en cada segmento, una identificación o seguimiento de los locutores, una transcripción automática de lo dicho en cada segmento y una conversión a lemas que facilite la recuperación.

El grupo GTTS desarrolló un sistema de indexado, inicialmente denominado *Ehiztari* pero posteriormente rebautizado como *Hearch*, que trataba de abordar todas estas cuestiones. La arquitectura del sistema de indexado multilingüe (daba cobertura al español, euskera e inglés) fue presentada inicialmente en [27, 29]. En [28] se describió la estructura XML del recurso que contenía toda la información de cada recurso multimedia, y que era enriquecida por cada fase de procesamiento del sistema, y en [186, 180] fueron descritas las diversas evoluciones con las que contó el desarrollo del sistema.

El desarrollo del sistema de indexado ha dado lugar también a algunas aportaciones específicas que no se enmarcan directamente en las líneas de investigación de las secciones anteriores. Así, por ejemplo, en [153] se mostró un sistema de segmentación acústica y seguimiento de locutores para contenidos multimedia, y en [157] se desarrolló un sistema de clasificación y segmentación acústica que fue presentado en la Albayzin 2010 Audio Segmentation Evaluation.

6.7. Alineamiento y subtitulado de sesiones parlamentarias

Como ya se ha comentado en la sección anterior, el sistema de indexación y búsqueda fue enfocado tanto a recursos de informativos televisivos como a sesiones parlamentarias. En el caso del Parlamento Vasco, además, se contaba con una información adicional: las actas parlamentarias. Las actas de las sesiones parlamentarias reflejan de manera formal el contenido de una sesión, pero no corresponden a una transcripción precisa del audio de dicha sesión. Por un lado, contienen ciertos elementos estructurales que no tienen por qué corresponder con el audio, como puedan ser las presentaciones de cada nuevo orador o las votaciones realizadas, que se muestran mediante una estructura estandarizada. Por otro lado, es común filtrar las disfluencias e incluso corregir lo dicho por el orador.

A raíz de los requerimientos legales sobre accesibilidad que afectan a las administraciones públicas desde el 31 de diciembre de 2008 (RD 1494/2007), el Parlamento vasco se vio obligado a incluir subtítulos en los vídeos de las sesiones parlamentarias, y el grupo GTTS comenzó a prestar dicho servicio, el cual se basó en el alineamiento temporal del audio de los vídeos y las actas de cada sesión. En [35] se presentó el método desarrollado para el alineamiento, que dadas las características del problema (transcripciones imprecisas y sesiones de hasta tres horas de duración) no podía ser abordado como un tradicional problema de alineamiento forzado. Dicho método se basaba en obtener una decodificación acústico-fonética del audio, y posteriormente alinear dicha secuencia de fonemas frente a una transcripción fonética de las actas parlamentarias. Posteriormente, el sistema de alineamiento fue evaluado y comparado frente a los resultados obtenidos en trabajos previos de otros autores, los cuales hacían uso de complejos sistemas de reconocimiento que implicaban un coste computacional considerablemente superior [38, 36]. Los resultados obtenidos demostraron que el método propuesto podía obtener resultados equiparables a los de aquellos. En [39] se presentaron algunas mejoras añadidas al sistema de alineamiento y subtitulado, tanto en cuanto a rendimiento como en cuanto a costes computacionales. Más recientemente, se han propuesto algunas mejoras en la fase de alineamiento de secuencias fonéticas [37]. Estos cambios tienen en cuenta la información previa sobre la caracterización de los errores cometidos por el decodificador acústico-fonético, y reflejan una mejora significativa en la calidad del resultado del alineamiento.

6.8. Búsqueda por voz en recursos de audio

La búsqueda por voz en recursos de audio, comúnmente denominada Query-by-Example Spoken Term Detection, trata de localizar segmentos contenidos en un conjunto de recursos de audio en base a una consulta de voz. Se trata de generar un ranking de todos los posibles segmentos de audio en función de la semejanza que estos puedan tener con la consulta realizada. El grado de abstracción de la semejanza puede ir desde una mera semejanza acústica, en la que no es preciso tener un conocimiento expreso de la lengua hablada, hasta una semejanza semántica, en la que sería preciso contar con un conocimiento profundo de la lengua. En algunos casos, no se requiere

localizar el segmento dentro del recurso, sino que solamente se desea saber en qué grado contiene ese recurso algún segmento que se asemeje a la consulta. En tal caso, la tarea puede formalizarse como una verificación de la consulta realizada, de manera similar a la verificación de la lengua o del locutor.

Al igual que en algunas de las líneas de investigación previamente citadas, los primeros desarrollos en el área de la búsqueda por voz en recursos de audio llevados a cabo por el grupo de investigación GTTS se debieron a la participación en sendas evaluaciones organizadas por la Red Temática en Tecnologías del Habla, organizadora de las evaluaciones Albayzin, y la iniciativa MediaEval, dedicada a evaluar nuevos algoritmos para el acceso y recuperación de información multimedia:

- En 2012 se participó en las evaluaciones MediaEval 2012 Spoken Web Search y Albayzin 2012 Search on Speech. Para ambas evaluaciones se desarrolló un sistema basado en decodificadores fonéticos que obtenía, por un lado, múltiples decodificaciones de la consulta realizada y, por otro lado, una red de fonemas a partir de cada recurso, para posteriormente localizar segmentos con similitud fonética. Dado que se basaba en un principio de similitud fonética, el sistema desarrollado tenía la peculiaridad de hacer uso de decodificadores fonéticos no coincidentes con la lengua o lenguas procesadas [182, 183].

- En 2013 se participó en la MediaEval 2013 Spoken Web Search. A diferencia del sistema desarrollado para las evaluaciones anteriores, este sistema también partía de la salida de un decodificador fonético pero en vez de utilizar secuencias o mallas de fonemas, tomaba directamente el vector de posteriores del decodificador. Dicho vector era utilizado como vector de parámetros, de tal manera que la semejanza entre la consulta realizada y los posibles segmentos del recurso de audio era obtenida mediante un alineamiento recursivo de ambas secuencias de parámetros [168, 170].

- En 2014 se participó en la MediaEval 2014 Query by Example Search on Speech. Los cambios más destacables frente al sistema anterior fueron el adaptarse a la nueva tarea, que no requería localizar los segmentos, sino únicamente verificar la existencia de la consulta, y el uso de un conjunto reducido de unidades fonéticas en base a su manera y lugar de articulación [169].

El aspirante también participó activamente en la organización de las evaluaciones MediaEval 2013 Spoken Web Search [20] y MediaEval 2014 Query-by-Example Speech Search Task [18]. Esta colaboración se centró sobre todo en la definición de las métricas de evaluación, donde se estudió la relación entre las métricas utilizadas para evaluar los sistemas de detección de términos y las utilizadas para evaluar los sistemas de verificación de locutor [152]. En este estudio se propuso además una nueva métrica de evaluación basada en la entropía cruzada que, en vez de basarse meramente en los errores de decisión del sistema de búsqueda, ofrecía la capacidad de medir la cantidad de información que aporta cada sistema. Esta métrica, una adaptación de otra métrica comúnmente utilizada en el área de la verificación de la lengua y el locutor, permite además calibrar sistemas heterogéneos, dando lugar a una sencilla fusión ponderada de sistemas, como pudo demostrarse en [15]. Con posterioridad a la MediaEval 2013

Spoken Web Search, se realizó un estudio de fusiones de los sistemas participantes que utilizó esta misma técnica de calibración y fusionado [19].

Capítulo 7

Conclusiones y Trabajo Futuro

7.1. Conclusiones

Sautrela es un entorno de desarrollo basado en plugins cuya finalidad es la generación de cadenas de procesamiento de información. La información, que puede ser representada mediante vectores numéricos o cadenas de texto, fluye a través de una cadena de procesamiento o *Engine* y es transformada de manera concurrente por cada uno de los *Procesadores* que la componen.

Un plugin contiene un conjunto de *Procesadores* y *Comandos* diseñados con un objetivo particular. Basándose en el modelo de componentes software JavaBeans™, Sautrela integra de forma natural tanto plugins propios como plugins desarrollados por terceros. Las *Engines* pueden ser parametrizadas, pudiendo fijar algunos de los parámetros de los *Procesadores* o permitir su determinación posterior, dando lugar a opciones de línea de comando configurables. Sautrela define además un mecanismo para la generación automática de documentación de las *Engines*, los *Comandos* y los *Procesadores*.

Sautrela incluye un conjunto de plugins que da cobertura a las tareas propias de las tecnologías del habla: la adquisición y procesamiento de señales de audio y el modelado. Cuenta además con otros plugins más genéricos, que implementan sencillas transformaciones o procesamientos de utilidad, cuya combinación puede dar lugar a complejas cadenas de procesamiento.

El plugin de modelado concentra gran parte de las contribuciones novedosas de Sautrela. Un reducido y sencillo conjunto de interfaces define la abstracción funcional de un autómata de estados finitos ponderado (WFSA). Dicha abstracción permite integrar diversos tipos de modelos diferentes, independientemente de si sus espacios de observación corresponden a distribuciones discretas o continuas, escalares o vectoriales. Todo modelo, propio o de terceros, que implemente la interfaz WFSA podrá ser utilizado en el entorno de modelado de Sautrela.

El entorno de modelado se Sautrela se basa en un mecanismo unificado de decodi-

ficación y re-estimación de modelos, y viene acompañado de sendos *Procesadores* que llevan a cabo dichas tareas. Todo modelo que implemente la interfaz WFSA podrá ser utilizado como fuente de conocimiento para decodificar una secuencia de observaciones de entrada y obtener la secuencia más probable de transiciones. Del mismo modo, todo modelo que implemente la interfaz WFSA podrá ser re-estimado en base al conjunto de secuencias de observaciones de la entrada. La re-estimación de modelos puede realizarse con el criterio de maximización de la verosimilitud (ML) o el de la maximización de la probabilidad a posteriori (MAP), en cuyo caso se implementa una versión restringida del método de re-estimación MAP, que reduce a un único parámetro intuitivo toda la información previa relativa al conjunto de parámetros del modelo.

El conjunto de modelos implementados en Sautrela incluye los denominados modelos de Markov por capas (LMM), que integran en un único WFSA distintos niveles o capas de conocimiento representados por conjuntos de WFSAs. Cada capa o conjunto de modelos, modela sus clases en función de las unidades de la capa inferior, de manera que existe una correspondencia directa entre las observaciones de los modelos de una capa y los modelos de la capa inferior. Este formalismo ofrece una excepcional flexibilidad a la hora de diseñar escenarios (sistemas) tanto de reconocimiento (decodificación) como de aprendizaje (re-estimación). Gran parte de los sistemas de reconocimiento pueden ser formalizados como la decodificación mediante un LMM y, del mismo modo, muchos de los mecanismos de supervisión a la hora de re-estimar conjuntos de modelos pueden ser formalizados como la re-estimación de un LMM. Además, la re-estimación de LMMs hace posible la re-estimación acoplada de diferentes capas de conocimiento, las cuales podrían haber sido estimadas por separado.

Los procesos de decodificación y re-estimación pueden hacer uso del método de búsqueda en haz (*beam search*), reduciendo simultáneamente los costes temporal y espacial requeridos por los correspondientes algoritmos. Para poder implementar métodos de búsqueda más complejos que requieran el conocimiento expreso de los modelos que toman parte en la decodificación, Sautrela cuenta con modelos específicos que implementan internamente el proceso equivalente de búsqueda. Tal es el caso de la búsqueda en árbol léxico, para la cual existe el modelo equivalente TreeModel, que integra en un único WFSA un modelo de lenguaje y un conjunto de modelos léxicos, y puede ser utilizado como cualquier otro WFSA: puede ser directamente utilizado en un proceso de decodificación, o puede ser integrado en un LMM que será posteriormente utilizado en un proceso de decodificación.

La decodificación y re-estimación unificada de modelos a través de interfaces es una de las principales aportaciones de Sautrela. La capacidad de interaccionar con un modelo y estimar la esperanza de sus transiciones sin necesidad de conocer su naturaleza interna es la clave de ambos mecanismos. Sin embargo, el desconocimiento del gobierno interno de un modelo conlleva ciertos retos a la hora de implementar ciertos mecanismos algorítmicos que incluyen los entornos de desarrollo tradicionales.

Los mecanismos unificados de decodificación y re-estimación de modelos, y el formalismo de los LMMs, junto a su capacidad de decodificación y re-estimación acoplada, conforman, sin lugar a dudas, el conjunto de aportaciones más relevantes del presente trabajo.

7.2. Trabajo futuro

Los siguientes apartados resumen el conjunto de posibles acciones futuras que quedan abiertas en el momento de finalizar la redacción de la presente memoria.

7.2.1. Sautrela como proyecto de software

Desde sus orígenes, Sautrela fue concebido como un proyecto de código libre. Sin embargo, debido a la falta de conocimiento en la gestión de software y al hecho de que ha contado prácticamente con un único desarrollador, no se ha prestado suficiente atención a su gestión como proyecto de software. Tomando como referencia los estándares actuales, Sautrela debería estar alojado en una plataforma de desarrollo colaborativo, abriéndolo a la comunidad y llevando un control de versiones más claro.

Todas las clases de Sautrela deberían contar también con la correspondiente prueba unitaria [214], que permita realizar las pruebas de regresión pertinentes en caso de realizar modificaciones de parte del código.

7.2.2. Java™ 1.8

La plataforma Java™, como conjunto formado por el propio lenguaje, la máquina virtual y la librería estándar de clases, es un ecosistema en constante evolución. Sautrela está escrito en Java y ha ido evolucionando y adaptándose a las nuevas características de esta plataforma. El último gran paso evolutivo de la plataforma, Java™ 1.8, vio la luz en Marzo de 2014, introduciendo por vez primera elementos de la programación funcional en el lenguaje de programación. Debido a la profundidad de los cambios y a su falta de implantación, Sautrela actualmente no incorporá ningún elemento propio de la versión 1.8 de Java. No obstante, resulta obvio que deberían ir adoptándose los cambios que repercutan en un código más claro y eficiente.

A continuación se da cuenta de algunos de los elementos novedosos de la última versión que se estima que podrían aportar mejorar sustanciales.

Expresiones Lambda

En la nueva versión de Java, los métodos pasan a ser “ciudadanos de primera”, de manera que pueden ser referenciados (previamente solo podían ser evaluados). Esta característica es fundamental para un enfoque funcional de la programación. Una expresión Lambda no es otra cosa que la declaración de una función anónima (en este contexto se tiende a utilizar el término *función* en vez de *método*) que tiende a ser utilizada en aquellas situaciones en las que no se cuente de antemano con una función que realice una tarea específica.

Un elemento interesante a tener en cuenta a la hora de plantear posibles cambios en el código de Sautrela es la introducción de las denominadas *interfaces funcionales* (una interfaz que contiene un único método), ya que pueden ser implementadas en línea mediante una expresión Lambda.

Flujos (Streams) y colecciones

El nuevo paquete `java.util.stream` proporciona una API que soporta la declaración funcional sobre flujos (*Streams*) de elementos. Esta API está integrada, a su vez, en la API Collections, permitiendo operaciones masivas sobre colecciones de elementos, tales como transformaciones map-reduce secuenciales o paralelas.

Los Streams de Java, que incluyen mecanismos automáticos de paralelización, son en cierta manera análogos a las engines de Sautrela. Sin embargo, la paralelización de los Streams difiere totalmente del procesamiento concurrente de los procesadores de Sautrela. En una engine, cada procesador representa un hilo independiente de procesamiento y el paralelismo se da, por lo tanto, a nivel de procesador. Sin embargo, los Streams de Java representan una cadena de funciones en la que el procesamiento ocurre a demanda de la última de las funciones, lo que genera el encadenamiento de las funciones de procesamiento anteriores. Al paralelizar un Stream, se paraleliza la cadena completa de procesamientos, manteniendo el esquema de procesamiento bajo demanda. En este sentido, no queda clara la posibilidad de migrar el sistema de procesamiento de engines a las nuevas Streams de Java (sin modificar la interfaz de los procesadores), pero, sin duda, se trata de una opción a considerar.

Anotaciones de tipo

Las anotaciones son elementos del lenguaje Java que permiten enriquecer (*decorar*, en terminología de patrones de diseño) los elementos de un programa con información adicional. En las versiones previas, solo era posible anotar las declaraciones, y eran comúnmente utilizadas para la generación de documentación y la configuración del compilador: detección de errores, supresión de avisos, etc. Actualmente las anotaciones pueden aportar cierto control sobre los tipos o valores de las referencias. No existe un conjunto oficial de anotaciones de tipo ni una herramienta oficial que las utilice, sino que se trata de una sintaxis oficial que permite que herramientas de terceros (como el framework Checker creado de la University of Washington [8]) realicen el chequeo de tipos. El uso de dichas anotaciones de tipo puede facilitar la depuración del código desarrollado, dado que permiten el chequeo de ciertas precondiciones sin necesidad de alterar el código fuente.

7.2.3. Audio

La librería estándar de clases da un soporte muy limitado a los formatos de audio. Tratándose de un entorno orientado a las tecnologías del habla, parece más que necesario tener acceso a un mayor conjunto de formatos de audio. La solución lógica sería hacer uso de alguna librería externa que permita la lectura de ficheros de audio en diversos formatos.

7.2.4. Librería de álgebra lineal

Muchos de los cómputos internos de los procesadores pueden ser expresados y realizados de manera eficiente mediante álgebra lineal. Sin embargo, ello implica la necesidad de contar con la correspondiente librería que dé soporte a las operaciones

más comunes. Lo más lógico sería adoptar alguna de las librerías de álgebra lineal más extendidas en Java, pero teniendo en cuenta que algunas de ellas se basan a su vez en otras librerías (BLAS/ATLAS), cabe considerar la conveniencia de adoptar una solución sin dependencias, entre las cuales se encuentran:

- EJML - Efficient Java Matrix Library [1].
- la4j - Linear Algebra for Java [3].
- OjAlgo - Open Source Java code to do mathematics, linear algebra and optimisation [5].
- Parallel Colt - a multithreaded version of Colt - a set of Open Source Libraries for High Performance Scientific and Technical Computing in Java [7].
- UJMP - Universal Java Matrix Package [14].

Todo ello supeditado, obviamente, a su rendimiento, que podría ser medido mediante el paquete Java Matrix Benchmark [2].

7.2.5. Procesamiento de señal

Lógicamente, el conjunto de procesadores que incluye Sautrela podría ser extendido con infinidad de nuevos componentes. A continuación nombraremos algunos procesadores que podrían aportar funcionalidades interesantes:

- **Transformación PCA**: Se trata de una sencilla transformación no supervisada que puede ser utilizada para reducir la dimensionalidad de los parámetros o simplemente para decorrelarlos [67], especialmente indicada si los modelos se basan en distribuciones Gaussianas con covarianza diagonal. Dado que se basa en la estimación de los autovectores de la matriz de autocorrelación, esta técnica sería sencilla de implementar si se contase con una librería de álgebra lineal.
- **Transformación LDA**: A diferencia de la transformación PCA, LDA requiere de una supervisión, ya que trata de estimar la transformación lineal que maximice la separabilidad de un conjunto predefinido de clases. No se trata, por tanto, de una sencilla transformación que pueda ser estimada directamente sobre un conjunto de secuencias de datos de entrada, ya que depende de la información de pertenencia de cada muestra al conjunto de clases a separar. Dichas clases pueden corresponder a clases acústicas (fonemas), identidades de locutor, etc. No obstante, podría implementarse un procesador suficientemente genérico que estime una transformación LDA bien en base a un conjunto de modelos, bien en base a una información de supervisión contenida en las cabeceras de los datos.
- **Normalización de tracto vocal**: Es una técnica de amplio espectro: la podemos encontrar tanto en sistemas de reconocimiento del habla como en sistemas de verificación de la lengua. No se trata de un procesador sencillo, ya que es el producto de diversas fases de procesamiento [102, 175]. Dado que Sautrela cuenta con todos los elementos necesarios, sería posible formalizar un mecanismo para implementarla.

- **Low-Variance Multitapers**: El producto de múltiples ventaneos ortogonales permite reducir la incertidumbre de estimación de la información espectral [148]. Esta técnica ha sido recientemente integrada en sistemas de verificación de locutor [93, 16], y su implementación no debería plantear complicación alguna.
- **Feature-space Maximum Likelihood Linear Regression (fMLLR)**: también denominada Constrained MLLR, se trata de una técnica de adaptación al locutor muy extendida en los sistemas de reconocimiento del habla. Consiste en estimar una transformación afín del espacio de parámetros que maximice la verosimilitud para cada uno de los locutores [103, 68, 222]. No obstante, esta estimación es totalmente dependiente del tipo de distribución que tengan los modelos acústicos, y normalmente es implementada bajo la premisa de distribuciones de mezclas de Gaussianas (GMMs). Aunque aparentemente parezca inviable, resultaría más que interesante investigar la posibilidad de formalizar un método de estimación que no dependa del conocimiento de la naturaleza interna de los modelos, implementando una transformación que pueda ser estimada simplemente a partir de la interfaz WFSA de los modelos.

7.2.6. Modelado

El módulo que contiene mayor número de aportaciones es también el módulo con mayor potencial de evolución. A continuación se resumen algunos aspectos que podrían ser mejorados o ampliados.

Decodificación y entrenamiento multihilo

Tanto HTK, Sphinx como Kaldi ofrecen la posibilidad de paralelizar el entrenamiento de los modelos acústicos, basándose en la ejecución simultánea de N entrenamientos independientes y la posterior fusión de las cuentas de entrenamiento acumuladas. Sautrela carece de una característica equivalente[1].

Ambos módulos de decodificación y entrenamiento podrían fácilmente paralelizar sus procesos internos, repartiendo el conjunto de datos entre N hilos, todos ellos trabajando frente a una única instancia del WFSA. Los cambios necesarios en el código serían mínimos. Sin embargo, ello no aseguraría una eficaz ejecución concurrente, que presentaría serios inconvenientes:

1. Como ya se comentó en la Sección 3.3, los mecanismos de exclusión mutua pueden provocar que el hipotético beneficio de una ejecución concurrente se diluya, debido a constantes bloqueos mutuos de los hilos de ejecución y al coste inherente del mecanismo de sincronización escogido.

2. Algunos autómatas hacen uso de una caché que acelera la respuesta a métodos ya evaluados previamente[2]. Pero si una misma instancia de autómata fuera accedida

[1] Los procesadores se ejecutan de manera concurrente, pero los procesos de decodificación y entrenamiento no implementan ningún mecanismo interno de paralelización.

[2] Por ejemplo, dado un instante, un mismo modelo acústico puede encontrarse en infinidad de contextos léxicos y de lenguaje diferentes. En tales casos, la consulta de la probabilidad acústica puede llegar a realizarse infinidad de veces, por lo que un mecanismo de cache que guarde el resultado de la última consulta acelerará el proceso de decodificación/entrenamiento.

simultáneamente por varios hilos, el mecanismo de caché resultaría ineficiente, por no decir perjudicial.

En decodificación, la solución más sencilla sería contar con tantas instancias del modelo como hilos de ejecución. En entrenamiento, sin embargo, no bastaría con tener instancias múltiples del modelo a estimar; sería preciso también contar con un mecanismo adicional de combinación de los estadísticos acumulados por cada una de las instancias, que, una vez finalizado el procesamiento de las secuencias de entrada, generase una única re-estimación. Partiendo de este análisis, se plantean dos posibles soluciones al problema:

- **Múltiples instancias independientes**: Se basaría en realizar N procesamientos independientes (haciendo uso de N instancias independientes o clones del WFSA). Como ya se ha comentado, para realizar la re-estimación del modelo sería necesario establecer un mecanismo de combinación. La siguiente modificación de la interfaz WFSA sería suficiente:

  - Los autómatas deberían implementar la interfaz `Cloneable`, que permitiría crear instancias de autómatas totalmente independientes.
  - El método de notificación de finalización de la re-estimación `dumpSuffStats()`, debería ser modificado para que fuera capaz de notificar la existencia de un conjunto de modelos que han sido re-estimados en paralelo. Su prototipo podría pasar a ser:

    ```
    void dumpSuffStats(WFSA[] all)
    ```

    donde el vector de modelos debería contener al conjunto completo de autómatas re-estimados en paralelo[3].

- **Múltiples instancias semi-independientes**: Incluso durante el entrenamiento, la mayor parte del tiempo se hace uso del subconjunto de métodos de decodificación, ya que para calcular las cuentas de entrenamiento es preciso realizar una decodificación forzada de la señal de entrada frente a una transcripción. Una vez obtenidos todos los posibles caminos, las esperanzas de las transiciones son transmitidas al modelo mediante la interfaz de entrenamiento. Existe, por tanto, la alternativa de generar instancias de autómatas semi-independientes, donde todos los elementos internos de los que depende la decodificación podrían ser replicados, mientras que los relativos al entrenamiento mantendrían una única instancia interna común (estableciendo mecanismos de exclusión mutua únicamente en los métodos de entrenamiento). La interfaz WFSA debería contar, por lo tanto, con un nuevo método:

  ```
  WFSA getTrainableClone();
  ```

[3] Este método es utilizado actualmente en Sautrela para entrenar GMMs en paralelo (Procesador GMMTrainer), pero no está implementado para los WFSAs.

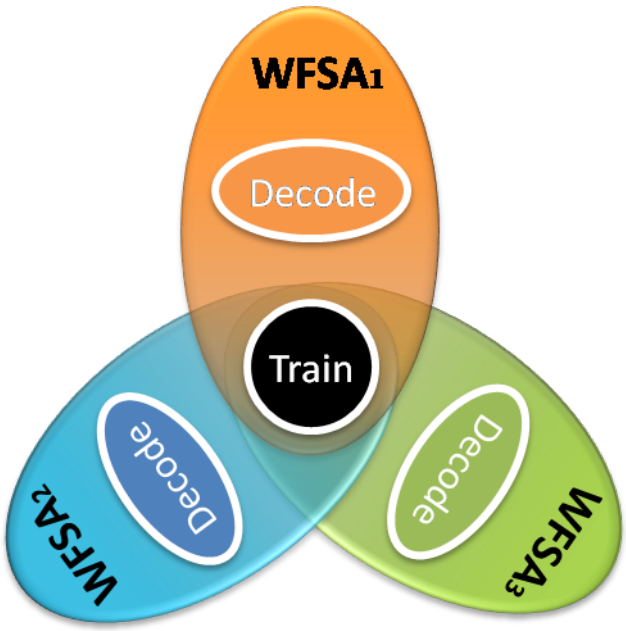


Figura 7.1 Representación de tres instancias de autómata compartiendo una misma sección de código de entrenamiento.

que dado un WFSA permitiese obtener otra instancia semi-independiente[4]. La Figura 7.1 muestra de forma gráfica tres modelos que comparten una misma estructura de entrenamiento, pero que en tiempo de decodificación se comportarán como instancias independientes.

Decodificación en tiempo real

El decodificador de Sautrela espera al final de la secuencia para iniciar la decodificación. En caso de hacer uso del decodificador con señales adquiridas en tiempo real, ello conlleva que el proceso de decodificación no comenzará hasta que termine la adquisición, lo que acarrea un notable retardo a la salida. Resultaría relativamente sencillo implementar un decodificador en tiempo real que permitiese reducir drásticamente el retardo de salida.

Re-estimación MMI

En base al teorema de transformaciones crecientes de funciones racionales [72], pueden obtenerse ecuaciones de re-estimación que maximicen el criterio de máxima información mutua (MMI, por sus siglas en inglés). El método que hace uso de estas

[4]También podría usarse directamente la interfaz `Cloneable`, aunque, como las instancias obtenidas no serían realmente independientes, no parece aconsejable.

ecuaciones es también denominado *Extended Baum-Welch*. La función objetivo de la información mutua es el cociente de dos verosimilitudes del conjunto de datos: la del numerador, condicionada al conocimiento de las clases que han generado las muestras, y la del denominador, sin información de supervisión alguna. El cociente de ambas verosimilitudes se relaciona con la reducción de entropía que supone el conocimiento de la identidad de las clases, y representa la información mutua: la cantidad de información (sobre el valor de las observaciones) que aporta el conocimiento de las identidades de clase, o equivalentemente, la cantidad de información (sobre la identidad de las clases) que aportan las observaciones.

Una característica muy interesante de las ecuaciones de re-estimación MMI es que su forma es muy parecida a una estimación MAP en la que tratamos de maximizar el numerador, pero a la que se le restan las aportaciones que de manera natural obtendríamos si tratáramos de maximizar el denominador. Podría interpretarse que se trata de un proceso de maximización donde aparecen esperanzas positivas (las relativas al numerador) y esperanzas negativas (las relativas al denominador), y la finalidad del término MAP no es otra que asegurar que la esperanza total acumulada de un elemento nunca sea negativa.

Esta reinterpretación de la re-estimación MMI abre las puertas al diseño de un mecanismo unificado de re-estimación MMI, si bien es verdad que a diferencia de la estimación MAP, donde el usuario podía establecer una esperanza previa acumulada cualquiera, en este caso existirá un valor mínimo desconocido a priori (dependiente del conjunto de datos) que deberá ser controlado y gestionado por dicho mecanismo.

Rediseño de la interfaz WFSA: emisiones asociadas a estados

La interfaz WFSA de Sautrela supone que las emisiones van asociadas a las transiciones. Esta decisión de diseño vino motivada por la necesidad de dar una cobertura más directa a los autómatas deterministas, "penalizando" en cierta manera a aquellos autómatas con emisiones asociadas a estados, como es el caso de los HMMs. Sin embargo, el formalismo de autómatas con emisiones asociadas a los estados permite desacoplar la topología de un modelo de las distribuciones de emisión, lo que favorece un mayor nivel de abstracción. El Listado 7.1 muestra una propuesta de rediseño de la interfaz WFSA, cuyas novedades podemos resumir en los siguientes puntos:

- Las emisiones están asociadas a los estados. Las transiciones representan un cambio de estado, pero no incluyen una observación (ni su correspondiente probabilidad).
- Todo autómata tiene un estado inicial, pero la primera emisión ocurrirá tras transitar del estado inicial a otro estado[5].
- Muchos de los métodos pasan a estar contenidos en los estados.
- Todos lo autómatas se presuponen no-deterministas. Los deterministas definen una sub-interfaz que contiene un método que nos indica cuál es la única transición posible dada una observación.

[5]El vector de probabilidades iniciales de un HMM representaría el conjunto de probabilidades de transición desde un estado origen adicional.

Listado 7.1 Rediseño de la interfaz de decodificación WFSA. Las emisiones quedan asociadas a los estados y desacopladas de las transiciones.

```
public interface Named {
   public String getName();
}

public interface State {
   public double getFinProb();
   public double getProb(Symbol y);
   public Transition[] getTrans();
   public Symbol[] getSymbols();
}

public interface Symbol extends Named {}

public interface Transition extends Named,Comparable {
   public State getSource();
   public State getDestination();
   public double getProbability();
}

public interface Alphabet extends Iterable {
   public Symbol valueOf(Object o);
}

public interface WFSA extends Named {
   public Alphabet getAlphabet();
   public State getInitState();
}

public interface DWFSA extends WFSA {
   public Transition getTrans(Symbol y);
}
```

Abstracción de las distribuciones de probabilidad y los modelos

Veíamos en la Sección 5.2 que las ecuaciones de re-estimación de los pesos de una mezcla de distribuciones no dependen de la naturaleza de dichas distribuciones. De igual manera, las ecuaciones de re-estimación de las probabilidades iniciales y de transición de un HMM no dependen de la naturaleza de las distribuciones de emisión asociadas a cada estado. Este hecho incide en la posibilidad de generar distribuciones y modelos genéricos cuya naturaleza quedará definida al ser instanciados, no de antemano.

Sautrela cuenta actualmente con una implementación de HMM discreto (emisiones con distribución categórica), otra de HMM continuo (emisiones con mezcla de Gaussianas), otra implementación de un GMM, etc. Sin embargo, sería posible aumentar el nivel de abstracción de los modelos y las distribuciones. Así, por ejemplo, podrían definirse:

- Una distribución categórica y una distribución Gaussiana.
- Una distribución de mezcla, compuesta por una distribución categórica y un conjunto de distribuciones cualesquiera.
- Un HMM : se trataría de un HMM genérico, como el de la Figura 5.5. Un HMM de K estados constaría de:
  - Probabilidades iniciales: una distribución categórica sobre los estados
  - Probabilidades de transición: para cada estado, una distribución categórica sobre el conjunto de estados.
  - Probabilidades de emisión: una distribución de probabilidad cualquiera por estado

En base a este formalismo, un GMM sería una mezcla de Gaussianas y un HMM continuo un HMM con distribuciones GMM. El código resultante sería más claro y conciso, ya que cada clase delegaría en otras gran parte del cómputo. El mero hecho de ampliar el conjunto de distribuciones, por ejemplo, a la familia exponencial (todas ellas pueden ser estimadas en base a estadísticos suficientes) abriría un amplio abanico de posibilidades de modelado.

No obstante, para poder instanciar este tipo de distribuciones o modelos a partir de una especificación XML (que es el formato de representación estándar de Sautrela) nos encontraríamos con un obstáculo a salvar. Pongamos el caso más sencillo de una distribución de mezcla. Si la mezcla puede contener cualquier conjunto de distribuciones de probabilidad, entonces el formato de su representación XML no puede ser prefijado de antemano (más allá de que debe contener una distribución categórica y un conjunto cualquiera de distribuciones) y el mecanismo de instanciación debe ser muy flexible. Este mismo problema ha sido abordado en la representación XML de los LMM, donde la solución adoptada es bien sencilla: no existe un único mecanismo de instanciación, sino que a la hora de instanciar objetos contenidos en ciertos elementos del XML, se delega dicha instanciación a otra clase. El Listado 7.2 muestra un ejemplo de lo que podría ser el XML que representa una mezcla. La clase

Listado 7.2 Fichero `DiagonalCovGMM.xml` con propuesta de instanciación delegada. Una mezcla de distribuciones podría estar compuesta por una distribución categórica y un conjunto cualquiera distribuciones. La instanciación de la distribución delegaría en cada una de las clases la instanciación de los elementos internos.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<Distribution className="MixtureDistribution">
   <Distribution className="CategoricalDistribution"> 0.7 0.2 0.1 </Distribution>
   <Distribution className="DiagonalGaussianDistribution">
      <Mean> 2.4 -0.8 </Mean>
      <Variance> 0.4 0.2 </Varinance>
   </Distribution>
   <Distribution className="DiagonalGaussianDistribution">
      <Mean> 10.8 2.5 </Mean>
      <Variance> 0.6 0.4 </Varinance>
   </Distribution>
   <Distribution className="DiagonalGaussianDistribution">
      <Mean> 6.4 -3.8 </Mean>
      <Variance> 1.2 2.1 </Varinance>
   </Distribution>
</Distribution>
```

`MixtureDistribution` sería la encargada de instanciar el objeto; sin embargo, delegaría en las clases `CategoricalDistribution` y `DiagonalGaussianDistribution` la instanciación de la distribución de pesos y el resto de distribuciones de cada componente (en este caso, Gaussianas con covarianza diagonal). El Listado 7.3 muestra un ejemplo similar en el que la mezcla está compuesta por Gaussianas con matriz de covarianza completa.

Modelos, estados o distribuciones ligadas

El uso de ligaduras es muy común en el modelado acústico basado en HMMs. La ligadura de un elemento no es otra cosa que establecer que existe una única instancia de dicho elemento que es compartida por diversos contenedores. Es muy común, por ejemplo, definir un amplio conjunto de unidades de tri-fonemas (unidades fonéticas con contexto a izquierda y derecha), pero establecer que los estados centrales de todas las variantes contextuales de un mismo fonema son idénticos. Ello permite la especialización de los modelos acústicos a los contextos definidos, manteniendo una zona de modelado central común.

Sautrela no incorpora ningún mecanismo de ligadura. No obstante, no resultaría complicado que las descripciones XML de los elementos (modelos, estados, distribuciones, etc.) incorporen un identificador único. El descriptor XML podría contener elementos vacíos que hagan referencia a algún elemento previamente instanciado. El Listado 7.4 muestra el ejemplo de un conjunto de modelos que contiene dos HMMs que podrían representar a los trifonemas `i_a_o` y `e_a_i`. Ambos modelos podrían compartir una única instancia de estado central.

Listado 7.3 Fichero `FullCovGMMGMM.xml` con propuesta de instanciación delegada. Una mezcla puede contener también componentes Gaussianas con matriz de covarianza completa.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<Distribution className="MixtureDistribution">
   <Distribution className="CategoricalDistribution"> 0.7 0.2 0.1 </Distribution>
   <Distribution className="GaussianDistribution">
      <Mean> 2.4 -0.8 </Mean>
      <Covariance>
         0.8 0.2
         0.2 0.7
      </Covariance>
   </Distribution>
   <Distribution className="GaussianDistribution">
      <Mean> 10.8 2.5 </Mean>
      <Covariance>
         0.9 0.3
         0.3 0.8
      </Covariance>
   </Distribution>
   <Distribution className="GaussianDistribution">
      <Mean> 6.4 -3.8 </Mean>
      <Covariance>
         0.6 0.4
         0.4 0.9
      </Covariance>
   </Distribution>
</Distribution>
```

Listado 7.4 Fichero `HMMSet.xml`. con propuesta de uso de identificadores en los descriptores XML. Una vez se ha definido un estado, puede ser referenciado más adelante. Ambos HMMs compartirán una única instancia del estado.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<WFSASet>
   ...
   <HMM name="i_a_o" className="HMM">
      ...
      <State id="1225">
         ...
      </State>
      ...
   </HMM>
   ...
   <HMM name="e_a_i" className="HMM">
      ...
      <State id="1225"/>
      ...
   </HMM>
   ...
<WFSASet>
```

TreeModels a partir de diccionarios ponderados

Actualmente, los diccionarios utilizados para la creación de TreeModels no permiten establecer pesos (probabilidades) a las transcripciones alternativas. La ponderación de las posibles alternativas mejoraría sensiblemente el modelado léxico. En ausencia de pesos, estos podrían presuponerse unitarios, obteniendo como resultado distribuciones equiprobables como las actuales.

TreeModels asociados a estados del modelo de lenguaje

En la implementación actual del TreeModel, existe una única instancia de árbol cuyas probabilidades de transición combinan las probabilidades de los modelos léxicos y las probabilidades de unigrama del modelo de lenguaje, lo que precisa del correspondiente ajuste de probabilidad del modelo de lenguaje al llegar al final de una palabra. Sería posible generar tantas instancias de árboles como estados tiene el modelo de lenguaje, de tal manera que las probabilidades de transición del árbol léxico fuesen las correctas en todo momento. Dichos árboles no tienen por qué ser generados previamente (de manera estática), sino que pueden ser generados bajo demanda, reduciendo el coste espacial, ya que la estimación de las probabilidades de transición del árbol es relativamente sencilla.

Serialización de objetos

Sautrela hace uso extenso del XML para la representación de objetos. Si bien es verdad que ello facilita la interpretación e incluso la edición de los descriptores, no es menos cierto que el tiempo requerido para instanciar complejos modelos a partir de un descriptor XML puede resultar excesivo. Java ofrece la posibilidad de la serialización, un sencillo mecanismo que permite el volcado y recuperación binaria de objetos. Podría formalizarse un sencillo mecanismo genérico que permitiese la instanciación de objetos serializados y su volcado a XML, y viceversa: la instanciación mediante XML y su serialización. Ello permitiría contar con un mecanismo más rápido de carga de modelos, sin necesidad de renunciar a una representación textual de la información.

Apéndice A

Sautrela: Instalación y casos de uso

A.1. Introducción

El presente apéndice ha sido diseñado con la finalidad de servir como manual de instalación y uso del entorno de desarrollo de Sautrela. Tomando como hilo conductor la creación de un Reconocedor Automático del Habla (RAH), se indican todos los pasos a seguir: desde la instalación del software, pasando por la preparación de los datos y el entrenamiento de modelos, y concluyendo con la ejecución y optimización del reconocedor. Durante cada una de las fases, se indican y comentan los diferentes elementos que conforman la arquitectura de Sautrela. El presente manual no asume familiaridad alguna del lector con las Tecnologías del Habla. No obstante, dicho conocimiento resulta crucial si se pretende utilizar Sautrela en contextos distintos al aquí desarrollado.

El resto del apéndice se organiza de la siguiente manera. La Sección A.2 describe los pasos a seguir para la correcta instalación del entorno Sautrela. La Sección A.3 realiza una breve introducción de la estructura de los sistemas de RAH. La Sección A.4 describe el software de transcripción grafema-fonema que será utilizado, y la Sección A.5 muestra los pasos a seguir para la creación de una base de datos acústica compatible con Sautrela a partir de un conjunto de grabaciones y sus correspondientes transcripciones ortográficas descargadas desde VoxForge. En la Sección A.6 se muestra el proceso de parametrización de las señales de audio, introduciendo para ello el empleo de *Engines* parametrizadas. La Sección A.7 muestra el proceso de creación y entrenamiento de los modelos acústicos, y la Sección A.8 se centra en el análisis de la calidad de las señales de entrenamiento, detectando y excluyendo aquellas que pudieran degradar el rendimiento del sistema. En las Secciones A.9 y A.10 se muestra el proceso de creación y estimación de los modelos léxicos y de lenguaje, respectivamente. Partiendo de los componentes anteriormente creados, la Sección A.11 describe cómo crear un único modelo unificado que integra los tres niveles de conocimiento que conforman un sistema de RAH: el fonético, el léxico y el de lenguaje. La Sección A.12

muestra el proceso de ejecución del reconocedor (proceso de decodificación). Finalmente, la Sección A.13 muestra los pasos a seguir para la creación de un sistema de RAH de gran vocabulario.

A.2. Instalación

Sautrela ha sido probado en entornos Windows, Mac OS X y Linux. A excepción de la presente sección de instalación, todos los ejemplos del presente apéndice presuponen un entorno de trabajo Unix/Linux con intérprete de comandos Bash.

Requisitos

- Java™1.7 o superior (puede descargar una versión actualizada desde http://www.java.com). Compruebe que versión instalada es la correcta (1.7 o superior)[1]:

```
$ <path_to_java> -version
java version "1.8.0_66"
Java(TM) SE Runtime Environment (build 1.8.0_66-b18)
Java HotSpot(TM) 64-Bit Server VM (build 25.66-b18, mixed mode)
```

Descarga de Sautrela

El entorno de desarrollo de Sautrela consiste en un único fichero JAR que puede ser descargado desde http://sautrela.org/lib/Sautrela.jar. Asimismo, es posible descargar el conjunto de ficheros utilizados en el presente apéndice desde: http://sautrela.org/lib/ApendixAFiles.tgz.

Configuración

Al ejecutar la línea de comando:

```
$ <path_to_java> -jar <path_to_Sautrela.jar>
```

debería mostrarse la ventana de inicio de la Figura A.1. Esta ventana da acceso a una serie de aplicaciones gráficas: dos aplicaciones que permiten consultar la documentación de los *Procesadores* y los *Comandos*, una aplicación que permite la edición visual de *Engines* y una última aplicación que muestra un conjunto de sencillas demos. Sin embargo, el presente apéndice hará uso de Sautrela mediante la línea de comando:

```
$ sautrela [OPTIONS] [<Command|Processor|Engine> [args ...]]
```

donde `sautrela` es un alias de:

[1] Si el directorio de instalación de Java ya está incluido en su *path*, no es preciso indicar la ruta completa del comando `java`; bastará con ejecutar `java -version`.

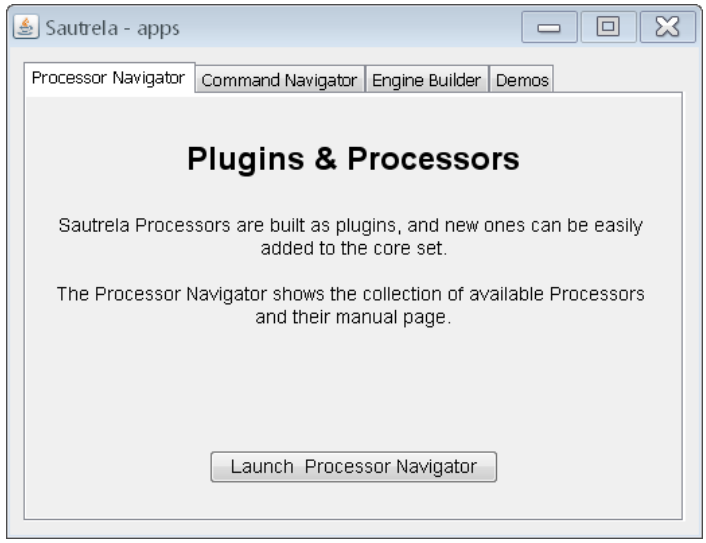


Figura A.1 Ventana de inicio de Sautrela.

```
<path_to_java> -jar <path_to_Sautrela.jar>
```

En función del sistema operativo, la instalación de Java™ y la localización del paquete `Sautrela.jar`, la definición de dicho alias variará. A continuación, se muestran tres ejemplos:

- En un entorno Unix/Linux que cuente con una instalación de Java™, se pueden añadir las siguientes líneas al fichero `~/.bashrc`[2] del usuario[3]:

```
function sautrela(){ <path_to_java> -jar <path_to_Sautrela.jar> "$@"; }
export -f sautrela
```

- En un entorno Windows que cuente con una instalación de Java™, se puede crear un archivo de ejecución por lotes `sautrela.bat` que debe encontrarse en alguno de los directorios incluidos en la variable de entorno `PATH` y que contenga:

```
@<path_to_java> -jar <path_to_Sautrela.jar> %*
```

- En un entorno Windows que cuente con PowerShell y una instalación de Java™, se puede configurar la carga automática de una función de manera análoga a lo dispuesto para los entornos Unix/Linux. Para ello, podemos añadir la siguiente

[2] En función de la configuración concreta del sistema, la elección del archivo `~/.bashrc` pudiera no ser adecuada, debiendo optar por otros archivos de configuración tales como `~/.profile` o `~/.bash_profile`.

[3] También es posible definir un alias: `alias sautrela='java -jar <path_to_Sautrela.jar>'`. Sin embargo, los aliases no son exportables, por lo que para poder utilizarlos en los scripts sería necesario volver a definirlos.

línea al script `Microsoft.PowerShell_profile.ps1`[4] que debe encontrarse en el directorio `%UserProfile%\Documents\WindowsPowerShell`:

```
function sautrela { <path_to_java> -jar <path_to_Sautrela.jar> $args }
```

Además, el usuario de Sautrela deberá tener en cuenta que, en el entorno PowerShell, la redirección de la salida estándar de un comando conlleva la recodificación a UTF-16 de su contenido. Ello es debido a que:

```
command > file.txt
```

es, en realidad, un alias de:

```
command | Out-File -FilePath file.txt
```

Y el *cmdlet* (comándulo) `Out-File` utiliza UTF-16 como codificación por defecto, de tal manera que, independientemente de la codificación del texto generado por el comando ejecutable[5], el fichero de salida `file.txt`. quedará codificado en UTF-16 [118]. Este comportamiento resulta especialmente pernicioso para aquellos comandos que vuelcan a la salida estándar código XML, ya que el prólogo del documento puede contener la declaración expresa de la codificación utilizada. Toda modificación de la codificación del documento XML debe ir acompañada de la modificación de dicha declaración; de lo contrario, se corrompe el documento. No obstante, el *cmdlet* `Out-File` permite hacer uso de una codificación alternativa, por lo que en tales casos deberá indicarse explícitamente la codificación a utilizar (a la hora de volcar documentos XML, Sautrela utiliza la codificación UTF-8):

```
command | Out-File -FilePath file.txt -Encoding utf8
```

A.3. Los Sistemas de Reconocimiento Automático del Habla

La percepción y comprensión del habla puede verse como el resultado de la interacción de diversos análisis que ocurren a varios niveles:

1. Nivel acústico: analiza las propiedades físicas propias de la señal de voz (energía, frecuencia, formantes, etc.).

2. Nivel fonético: analiza las unidades acústicas fundamentales (fonemas, pausas, etc.).

[4] En caso de encontrarse deshabilitada la ejecución de scripts (configuración de seguridad por defecto), esta puede activarse ejecutando, en modo administrador, el comando `Set-ExecutionPolicy Unrestricted`.

[5] PowerShell es un intérprete de comandos orientado a objetos que únicamente es compatible con comandos ejecutables cuya entrada o salida corresponda bien con objetos propios de la plataforma .NET, bien con texto. En otras palabras, son incompatibles los comandos ejecutables que vuelcan a la salida un flujo binario de datos.

3. Nivel léxico: analiza la combinación de las unidades del nivel fonético, para dar lugar a unidades léxicas (palabras).

4. Nivel sintáctico: analiza la combinación de las unidades léxicas, para dar lugar a frases acordes con la gramática de una lengua.

Estos cuatro niveles suelen integrarse en la arquitectura de un reconocedor como se muestra en las siguientes secciones. El nivel acústico es implementado mediante la fase de parametrización (*front-end*) del reconocedor. Se trata de una fase desacoplada del resto de niveles que trata de extraer de la señal de audio un conjunto de parámetros de bajo nivel. Estos parámetros deberán reflejar aquellas propiedades de interés para el reconocimiento posterior, y, a la vez, filtrar las propiedades irrelevantes (variabilidad de canal, variabilidad de locutor, etc.).

El resto de niveles suelen ser implementados mediante modelos estadísticos que son integrados en un motor de búsqueda, el cual tratará de obtener la frase pronunciada a partir de la información proveniente de la señal parametrizada y de cada uno de los niveles de conocimiento. Más concretamente, el nivel fonético suele ser implementado mediante modelos acústicos (típicamente Modelos Ocultos de Markov), el nivel léxico suele ser implementado mediante modelos léxicos (típicamente autómatas de estados finitos deterministas), y el nivel sintáctico se implementa mediante un modelo de lenguaje (típicamente n-gramas).

Algunos de los modelos mencionados requieren ingentes cantidades de datos para ser estimados de manera robusta. Por ejemplo, los modelos acústicos suelen precisar de grandes cantidades de audio (una base de datos acústica), y los modelos de lenguaje suelen precisar también de grandes cantidades de texto (un corpus de texto). Por el contrario, los modelos léxicos suelen generarse a partir de transcriptores basados en reglas o diccionarios de pronunciación generados por expertos.

Las arquitecturas de los reconocedores pueden clasificarse en base al esquema de integración que implementan [108]. Por un lado, están los sistemas no integrados, compuestos por módulos que hacen uso, cada uno de ellos, de un subconjunto de los niveles de conocimiento disponibles. Un ejemplo típico son los sistemas que en una primera fase, y basándose únicamente en el conocimiento acústico, obtienen una red o malla de fonemas, a partir de la cual obtener, en una segunda fase, la secuencia de palabras más probable (haciendo uso, ahora sí, de la información disponible en los niveles léxico y de lenguaje). Por otro lado, están los sistemas integrados, los cuales hacen uso simultáneo de la información disponible en todos los niveles durante el proceso de búsqueda de la secuencia de palabras más probable. Concretamente, los sistemas desarrollados en el presente apéndice se basarán en una arquitectura integrada.

A.4. El transcriptor fonético

La transcripción fonética nos permite representar el habla como una secuencia de símbolos, cuyo alfabeto se compone de los elementos de articulación mínima de los sonidos vocálicos y consonánticos de una lengua. Este alfabeto fonético corresponde al conjunto de modelos acústicos que pretendemos estimar. Durante el proceso de entrenamiento de los modelos acústicos, lo más común es partir de la transcripción

ortográfica de un conjunto de señales grabadas, bien porque hayan sido manualmente transcritas, bien porque correspondan a frases leídas. En tal caso, es preciso contar con un transcriptor grafema-fonema (en adelante transcriptor fonético).

Cuando se utiliza un método de transcripción grafema-fonema (sea mediante una herramienta automática o directamente por un experto), debe tenerse en cuenta que el resultado de una conversión grafema-fonema no tiene por qué coincidir con la pronunciación real. Por un lado existen variabilidades fonéticas de origen dialectal (diatópicas) y sociológico (diastráticas), y por otro lado, en ciertas lenguas puede darse el caso de que palabras con diferente pronunciación tengan una misma forma escrita. Por lo tanto, un transcriptor fonético tratará de obtener la transcripción fonética más probable (también denominada transcripción canónica).

En el presente apéndice se hace uso de un sencillo transcriptor fonético para el castellano, `ehu_transcribe`, que puede encontrarse junto con el resto de ficheros descargados (`00-database/scripts/ehu_transcribe.LATIN1.pl`):

```
$ ./ehu_transcribe.LATIN1.pl -h
Usage: ehutranscribe [file|-|<file] [-w] [-h]
  file  contains the sentences
  -     represents the standard input
  <file reads file though the standard input
  -w    word by word transcription
  -h    shows this help message
Units set: .ptkbdgmnhfzsxyclr@ieaou
          (/ll/ and /y/ are unified; /./ is a silence)
```

Puede verse cómo un mismo carácter `r` genera dos fonemas diferentes (representados mediante los símbolos `@` y `r`), la `h` no genera transcripción alguna, o la `ch` es representada por el símbolo `c`:

```
$ echo "Tengo un ratón y un loro" | ./ehu_transcribe.LATIN1.pl -w
tengo un @aton i un loro
$ echo "Carlos, te he echado de menos" | ./ehu_transcribe.LATIN1.pl -w
karlos te e ecado de menos
```

A.5. La base de datos acústica

Una base de datos acústica está compuesta por un conjunto de señales de audio transcritas. Para poder entrenar modelos acústicos mediante una base de datos acústica, será necesario contar con una transcripción a nivel fonético, y si se desea evaluar el reconocedor a nivel léxico (obtener una tasa de acierto de palabras), se necesitará también una transcripción a nivel léxico. Si las señales de audio provienen de habla leída (circunstancia bastante común), ya contamos con la transcripción léxica (las frases leídas, si suponemos que no ha habido errores de lectura), mientras que la transcripción fonética correspondiente podrá obtenerse mediante un transcriptor fonético.

Listado A.1 Fichero `exampleAcousticDataBase.xml` - Descriptor XML de una base de datos acústica. Cada recurso lleva asociado un nombre, un localizador y un conjunto de propiedades.

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE AcousticDataBase SYSTEM "http://sautrela.org/lib/AcousticDataBase.dtd">
<AcousticDataBase name="Voxforge Spanish 16KHz-16bit DataBase">
  <Resource name="train_00001">
    <Audio url="wav/AGOs-20100831-jdm/wav/es-0015.wav"/>
    <Property name="Ortho" value="de unas parras artificiales cuyas hojas parecían
        retazos de terciopelo"/>
    <Property name="Phon" value="d e u n a s p a @ a s a r t i f i z i a l e s k u
        y a s o x a s p a r e z i a n @ e t a z o s d e t e r z i o p e l o"/>
  </Resource>
  <Resource name="train_00002">
    <Audio url="wav/AGOs-20100831-jdm/wav/es-0016.wav"/>
    <Property name="Ortho" value="sillones de floreada cretona en torno de las
        mesas de bambú formaban islas"/>
    <Property name="Phon" value="s i y o n e s d e f l o r e a d a k r e t o n a e
        n t o r n o d e l a s m e s a s d e b a m b u f o r m a b a n i s l a s"/>
  </Resource>
  <Resource name="train_00003">
    <Audio url="wav/AGOs-20100831-jdm/wav/es-0017.wav"/>
    <Property name="Ortho" value="a las que se acogían grupos de personas para
        embadurnar"/>
    <Property name="Phon" value="a l a s k e s e a k o x i a n g r u p o s d e p e
        r s o n a s p a r a e m b a d u r n a r"/>
  </Resource>
  <Resource name="train_00004">
    <Audio url="wav/AGOs-20100831-jdm/wav/es-0018.wav"/>
    <Property name="Ortho" value="manteca y mermeladas el pan tostado husmear el
        perfume del té o seguir el burbujeo de las aguas minerales teñidas de
        jarabes y licores"/>
    <Property name="Phon" value="m a n t e k a i m e r m e l a d a s e l p a n t o
        s t a d o u s m e a r e l p e r f u m e d e l t e o s e g i r e l b u r b
        u x e o d e l a s a g u a s m i n e r a l e s t e h i d a s d e x a r a b
        e s i l i k o r e s"/>
  </Resource>
  <Resource name="train_00005">
    <Audio url="wav/AGOs-20100831-jdm/wav/es-0019.wav"/>
    <Property name="Ortho" value="camareros rubios de corta chaqueta azul y botones
         dorados pasaban con la bandeja en alto por los canalizos de este
        archipiélago humano"/>
    <Property name="Phon" value="k a m a r e r o s @ u b i o s d e k o r t a c a k
        e t a a z u l i b o t o n e s d o r a d o s p a s a b a n k o n l a b a n
        d e x a e n a l t o p o r l o s k a n a l i z o s d e e s t e a r c i p i
        e l a g o u m a n o"/>
  </Resource>
  ...
  ...
</AcousticDataBase>
```

Sautrela utiliza un documento XML para definir el conjunto de señales y transcripciones que forman una base de datos acústica (véase Listado A.1). Dicho descriptor está compuesto por un conjunto de recursos de audio (`Resource`). Cada recurso consta a su vez de un identificador (`name`), un localizador del recurso de audio (`Audio`) y un conjunto de propiedades (`Property`), y cada propiedad consta de un nombre y un valor. La finalidad de las propiedades es añadir información adicional a las señales de audio. El nombre de las propiedades puede ser cualquiera, pero en nuestro caso precisamos que exista al menos una propiedad que contenga la transcripción ortográfica (en el ejemplo del Listado A.1, la propiedad denominada `Ortho`) y otra que contenga la transcripción fonética (en el ejemplo del Listado A.1, la propiedad denominada `Phon`). La primera nos servirá para medir el rendimiento del reconocedor a nivel de palabra, mientras que la segunda la utilizaremos para entrenar los modelos acústicos y evaluar su bondad. Las transcripciones pueden utilizar cualquier carácter separador y los fonemas pueden estar definidos por cadenas de longitud variable. El uso de ficheros XML permite, además, manejar de manera adecuada diferentes codificaciones de texto, eliminando todos los errores derivados de inconsistencias en la codificación.

Las señales que forman una base de datos suelen estar divididas en subconjuntos, de manera que cada subconjunto pueda ser utilizado para diferentes fases (entrenamiento de modelos acústicos, evaluación de modelos acústicos, evaluación de reconocedor, etc.). El nombre de cada recurso es utilizado en Sautrela como identificador único, y los módulos que trabajan frente a bases de datos pueden seleccionar recursos mediante expresiones regulares, que deben coincidir con dichos identificadores. Por ello, es conveniente que el nombre contenga información del conjunto al que pertenece la frase. En el ejemplo del Listado A.1, por ejemplo, los nombres de las frases del subconjunto de entrenamiento acústico están precedidas por `train`, mientras que las correspondientes a los subconjuntos de evaluación podrían utilizar un prefijo diferente.

VoxForge: bases de datos acústicas libres

Uno de los problemas al que nos enfrentamos al tratar de entrenar modelos fonéticos es el acceso a una base de datos acústica. Aunque existe una gran cantidad de bases de datos acústicas en el mercado, la gran mayoría tiene costes elevados. VoxForge (`www.voxforge.org`) es un proyecto cuyo objetivo es construir de forma colaborativa bases de datos acústicas que puedan ser usadas por herramientas de reconocimiento de voz libres y de Código Abierto (Open Source). Aunque, como se comentará más adelante, la calidad de las grabaciones no es comparable a la de una base de datos comercial, su tamaño y la variabilidad de locutores pueden considerarse suficientes.

A continuación se muestran los pasos a seguir para crear una base de datos acústica en español, compatible con Sautrela, a partir de los recursos disponibles en el proyecto VoxForge.

Descarga y acondicionamiento

Desde la web de VoxForge podemos descargar las grabaciones donadas por usuarios. Dichas grabaciones se componen de señales de audio, transcripciones e información relativa al locutor. El Listado A.2 muestra un script que automatiza la descarga de

Listado A.2 `00-database/scripts/getVoxForge.sh` - Script de descarga de la base de datos acústica en español de VoxForge.

```
#!/usr/bin/env bash

# Download spanish database (16kHz,16bit) from voxforge

source scripts/config
mkdir data
wget --no-verbose -Lrl 1 -A "*.tgz" -P data -np -nd \
  http://www.repository.voxforge1.org/downloads/es/Trunk/Audio/Main/16kHz_16bit/
for f in data/*.tgz ; do
   echo $f
   tar -xzc data -f $f
   rm $f
done
```

todas las grabaciones en español.

El conjunto descargado[6] tiene un tamaño aproximado de 5 Gbytes y contiene en torno a 51 horas de grabación y más de 460 locutores diferentes[7]. El número de horas y la cantidad de locutores son adecuados para el entrenamiento de modelos acústicos. Sin embargo, hay algunos factores negativos que seguramente afectarán a la calidad de los modelos entrenados:

1. Los locutores no están equilibrados por género (el 80 % son hombres). Dado que ambos géneros tienen características acústicas diferenciadas, es conveniente que el conjunto de entrenamiento contenga un número suficientemente grande de cada género. La poca cantidad de mujeres conllevará sin duda una cobertura insuficiente de este conjunto de hablantes, y, por tanto, modelos poco robustos.

2. La procedencia de los locutores es muy amplia. Las grabaciones contienen una gran cantidad de variedades dialectales del español, con hablantes españoles y latinoamericanos. En principio, parece que la riqueza dialectal debería ser un factor positivo, pero solo lo será si el conjunto de locutores es lo bastante grande.

3. No consta una supervisión de las grabaciones. Se ha podido comprobar que algunas grabaciones no coincidían en absoluto con su transcripción, e incluso se han detectado señales que no contienen voz alguna. Una señal mal transcrita degradará los modelos acústicos si es utilizada para entrenarlos, o degradará artificialmente los resultados si es utilizada para evaluar el sistema.

4. Las condiciones de grabación son muy diversas, y muchas de las grabaciones contienen un ruido de canal considerable.

[6] Valores obtenidos a 31/10/2015. Al tratarse de un proyecto activo, la cantidad de datos disponible en VoxForge aumenta con el paso del tiempo.

[7] Debido a que los donantes pueden ser anónimos, no es posible saber con exactitud el número de locutores (la base de datos contiene más de 2000 envíos anónimos). No obstante, el número aportado considera todas las donaciones anónimas como un mismo locutor, por lo que cabría esperar que el número real de locutores sea mayor de 460.

Listado A.3 `00-database/scripts/normalize.sh` - Script de normalización de la base de datos acústica en español de VoxForge. Todas las señales quedan normalizadas a 1dB respecto del máximo.

```
#!/usr/bin/env bash

# Normalize all the database audio files to -1dBFS level

source scripts/config
SRC=data
DST=wav
# normalizing dB-level (below 0)
LEVEL=-1


mkdir $DST
for user in $(ls $SRC) ; do
   echo "Normalizing audio files from $user"
   mkdir $DST/$user
   for wav in $(ls $SRC/$user/wav) ; do
      sox --norm=$LEVEL $SRC/$user/wav/$wav $DST/$user/$wav
   done
done
```

Una vez descargada la base de datos, es preciso adecuar el volumen de las señales para que tengan un nivel similar. El script del Listado A.3 normaliza las señales de tal manera que queden un decibelio por debajo del nivel máximo de volumen.

Creación del vocabulario transcrito

Dado que las transcripciones de las señales de VoxForge son ortográficas, debemos obtener la transcripción fonética de cada señal. Para ello, y haciendo uso del transcriptor fonético `ehu_transcribe`, se creará un fichero que contenga la transcripción fonética de todas las palabras que aparezcan en las locuciones grabadas. El Listado A.4 contiene el código de un script que, en primer lugar, extrae el vocabulario de las transcripciones ortográficas, y a continuación, compone el vocabulario transcrito. El fichero resultante (`voc.transc`) contiene la transcripción fonética de cada una de las palabras que aparecen en las transcripciones ortográficas originales de VoxForge:

```
$ head voc.transc
a a
abandonar a b a n d o n a r
abaratamiento a b a r a t a m i e n t o
abarca a b a r k a
abierto a b i e r t o
abrir a b r i r
abstracción a b s t r a k z i o n
abstracto a b s t r a k t o
accesibilidad a k z e s i b i l i d a d
accionariado a k z i o n a r i a d o
```

Listado A.4 `00-database/scripts/createTranscVocabulary.sh` - Script de creación del vocabulario fonéticamente transcrito de la base de datos acústica. El fichero resultante contiene la transcripción fonética de cada una de las palabras que aparecen en las transcripciones ortográficas originales de VoxForge.

```
#!/usr/bin/env bash

# Create a phonetically transcribed vocabulary based on all the prompts

source scripts/config
SRC=data
# remove non alphabetic symbols and convert uppercase to lowercase
function cleanText { sed -e 's/[^[:alpha:] ]//g' -e 's/./\L&/g' ; }
# get phonetic transcription from UTF-8 encoded prompts
function ortho2phon {
   iconv -f UTF-8 -t LATIN1 \
     | ./scripts/ehu_transcribe.LATIN1.pl -w \
     | iconv -f LATIN1 -t UTF-8
}
# split all the character: "inputtext" --> " i n p u t t e x t"
function splitWord { sed -e "s/./ &/g" ; }


echo "Extracting promts sentences"
for f in $SRC/*/etc/prompts-original ; do cat $f ; echo "" ; done \
  | cut -d " " -f 2- | cleanText | tr " " "\12" | awk 'NF==1' | sort | uniq > voc

echo "Transcribing the vocabulary"
cat voc | ortho2phon | splitWord | paste -d "" voc - > voc.transc
rm voc
```

Definición de los subconjuntos de entrenamiento y validación

Una base de datos destinada al entrenamiento de modelos acústicos debería contar al menos con dos subconjuntos:

- Subconjunto de entrenamiento de modelos acústicos. Debería contener locuciones fonéticamente ricas y balanceadas. Es decir, un número suficiente de apariciones de cada uno de los fonemas en diferentes contextos fonéticos. Como se ha dicho previamente, también es deseable contar con el mayor número posible de locutores, balanceados también por género.
- Subconjunto de validación de modelos acústicos. No debería contener ni frases ni locutores que estén contenidos en el subconjunto de entrenamiento.

A la hora de diseñar esta partición, la base de datos de VoxForge ofrece directamente los siguientes dos subconjuntos:

- Subconjunto 1 (identificadores de frase `es` − `0000` a `es` − `0042`): 43 frases (locuciones a partir de un mismo texto) con más de 440 envíos por frase.
- Subconjunto 2 (identificadores de frase `01` a `140`): 140 frases con 9 envíos por frase.

Listado A.5 `00-database/scripts/createTrainValid.sh` - Script de generación de los índices de entrenamiento y validación. Cada línea de un fichero índice contiene un primer campo con el localizador del recurso de audio seguido de la transcripción de la frase.

```
#!/usr/bin/env bash

# Create train/validation partition maps (indexes)

source scripts/config
SRC=data
DST=wav
# remove non alphabetic symbols and convert uppercase to lowercase
function cleanText { sed -e 's/[^[:alpha:] ]//g' -e 's/./\L&/g' ; }
# clean text after first field
function cleanPrompt {
   mkfifo pipe
   tee >(cut -d " " -f 1 > pipe) | cut -d " " -f 2- \
     | cleanText | paste -d " " pipe -
   rm pipe
}


rm -f train.idx valid.idx
for user in $(ls $SRC) ; do
   prompts="$SRC/$user/etc/prompts-original"
   awk 'NF>1' $prompts | cleanPrompt | while read id txt ; do
      f="$DST/$user/$id.wav"
      if [ ! -f $f ] ; then
         echo " Warning: File $f not found, skipping" > /dev/stderr
         continue
      fi
      if [[ $id =~ ^es-0... ]] ; then
         echo $f $txt >> train.idx
      else
         echo $f $txt >> valid.idx
      fi
   done
done
```

Debido a la cantidad de frases y envíos, y si se pretende que los conjuntos de frases de entrenamiento y validación sean disjuntos, parece sensato utilizar el primer subconjunto para el entrenamiento de los modelos acústicos y el segundo para la validación. El Listado A.5 contiene el código de un script que obtiene los ficheros índice de cada subconjunto (`train.idx` y `valid.idx`). Cada fichero índice está compuesto por filas, cuyo primer campo indica el localizador del recurso de audio, seguido de la transcripción ortográfica (en minúsculas y sin signos de puntuación) de la frase locutada:

```
$ head -2 train.idx && tail -2 valid.idx
wav/abarzuaf-20101026-caf/es-0027.wav media docena de músicos uniformados
    lo mismo que los camareros agrupábanse sobre una tarima alrededor de
    un piano de cola
wav/abarzuaf-20101026-caf/es-0028.wav sus cabezas rubias de germanos y los
     arcos de sus violines destacábanse sobre los rectángulos luminosos de
```

```
     cuatro ventanas
wav/ubanov_es_01/133.wav sus medidas son noventa ochenta noventa
wav/ubanov_es_01/134.wav esta es la frase número cien o ciento setenta
```

Creación del descriptor XML de la Base de Datos Acústica

Una vez obtenido el diccionario fonético y generados los ficheros índice de los subconjuntos de entrenamiento y validación (que incluyen sus correspondientes transcripciones ortográficas), estamos en disposición de crear el documento XML que describirá una base de datos acústica compatible con Sautrela. El Listado A.6 muestra un script que crea el fichero descriptor `main.xml`. Cada uno de los recursos de audio lleva asociadas dos propiedades: una transcripción ortográfica y otra fonética; por otra parte, los identificadores de recurso son creados a partir del nombre del fichero índice de cada subconjunto, al cual se le añade un índice representado por cinco dígitos:

```
$ head -13 main.xml && tail -11 main.xml
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE AcousticDataBase SYSTEM "http://sautrela.org/lib/
    AcousticDataBase.dtd">
<AcousticDataBase name="Voxforge Spanish 16KHz-16bit DataBase">
  <Resource name="train_00001">
    <Audio url="wav/abarzuaf-20101026-caf/es-0027.wav"/>
    <Property name="Ortho" value="media docena de músicos uniformados lo
    mismo que los camareros agrupábanse sobre una tarima alrededor de un
    piano de cola"/>
    <Property name="Phon" value="m e d i a d o z e n a d e m u s i k o s u
     n i f o r m a d o s l o m i s m o k e l o s k a m a r e r o s a g r u
     p a b a n s e s o b r e u n a t a r i m a a l @ e d e d o r d e u n p
     i a n o d e k o l a"/>
  </Resource>
  <Resource name="train_00002">
    <Audio url="wav/abarzuaf-20101026-caf/es-0028.wav"/>
    <Property name="Ortho" value="sus cabezas rubias de germanos y los
    arcos de sus violines destacábanse sobre los rectángulos luminosos de
    cuatro ventanas"/>
    <Property name="Phon" value="s u s k a b e z a s @ u b i a s d e x e r
     m a n o s i l o s a r k o s d e s u s b i o l i n e s d e s t a k a b
     a n s e s o b r e l o s @ e k t a n g u l o s l u m i n o s o s d e k
     u a t r o b e n t a n a s"/>
  </Resource>
  <Resource name="valid_01253">
    <Audio url="wav/ubanov_es_01/133.wav"/>
    <Property name="Ortho" value="sus medidas son noventa ochenta noventa
    "/>
    <Property name="Phon" value="s u s m e d i d a s s o n n o b e n t a o
     c e n t a n o b e n t a"/>
  </Resource>
  <Resource name="valid_01254">
```

Listado A.6 `00-database/scripts/createAcousticDataBase.sh` - Script de generación del documento XML descriptor de la Base de Datos Acústica.

```
#!/usr/bin/env bash

# Create an AcousticDataBase XML document from index files
# Sentence names are created from the index file basename

source scripts/config

awk '
   BEGIN {
      print "<?xml version=\"1.0\" encoding=\"utf-8\" ?>"
      print "<!DOCTYPE AcousticDataBase SYSTEM \"http://sautrela.org/lib/
           AcousticDataBase.dtd\">"
      print "<AcousticDataBase name=\"Voxforge Spanish 16KHz-16bit DataBase\">"
   }
   ARGIND==1 {w=$1;$1="";tr[w]=$0}
   ARGIND>1 {
      if (FNR==1) {split(FILENAME,v,".");name=v[1]}
      url=$1;$1="";ortho=$0
      phon=""
      for (i=2;i<=NF;i++){
         if (! ($i in tr)) {
            print "ERROR: \""$i"\" not in dictionary" > "/dev/stderr"
            exit 1
         } else phon=(phon tr[$i])
      }
      number = sprintf("%05d",FNR);
      print " <Resource name=\"" name "_" number "\">"
      print " <Audio url=\"" url "\"/>"
      print " <Property name=\"Ortho\" value=\"" substr(ortho,2) "\"/>"
      print " <Property name=\"Phon\" value=\"" substr(phon,2) "\"/>"
      print " </Resource>"
   }
   END {
      print "</AcousticDataBase>"
   }
' voc.transc train.map valid.map > main.xml
```

```
    <Audio url="wav/ubanov_es_01/134.wav"/>
    <Property name="Ortho" value="esta es la frase número cien o ciento
    setenta"/>
    <Property name="Phon" value="e s t a e s l a f r a s e n u m e r o z i
     e n o z i e n t o s e t e n t a"/>
  </Resource>
</AcousticDataBase>
```

Análisis del contenido de la Base de Datos Acústica

El comando `AcousticDataBase` permite chequear el contenido de una base de datos acústica. En los siguientes ejemplos se comprueba, en primer lugar, la integridad de

la base de datos (que se tenga acceso a todos los recursos de audio declarados), y posteriormente, se obtiene el número de recursos que componen los subconjuntos de entrenamiento y validación (21621 y 1254, respectivamente):

```
$ sautrela -help AcousticDataBase

  AcousticDataBase (edu.gtts.sautrela.db.AcousticDataBase)

    show content of an AcousticDataBase

  Syntax:  AcousticDataBase  [-cnu] [-e enc] [-p list] < -i URL | -r
    RegExp > URL

    -c  Check (try opening) Audio/HTK Resources
    -n  Don't show Resource name
    -u  Don't show Resource URLs
    -e enc  Output encoding (default: UTF-8)
    -p list  A comma separated list of property names (default:ALL)
    -i URL  Locator of a Resource index (one ID per line)
    -r RegExp  Regular expression for Resource selection
    URL  Locator of an AcousticDataBase

$ sautrela AcousticDataBase -cnup "" -r ".*" main.xml
$ sautrela AcousticDataBase -up "" -r "train_.*" main.xml  | wc -l
21621
$ sautrela AcousticDataBase -up "" -r "valid_.*" main.xml  | wc -l
1254
```

Resumen

El Listado A.7 muestra un script que incuye todos los pasos descritos para la creación de una Base de Datos Acústica compatible con Sautrela a partir de las grabaciones contenidas en VoxForge. La salida que se vería al ejecutarlo sería:

```
$ ./run.sh
#######  1 - Download spanish VoxForge audio files  #######
2015-11-07  15:11:06 URL:http://www.repository.voxforge1.org/downloads/es/
    Trunk/Audio/Main/16kHz_16bit/AGOs-20100831-jdm.tgz [1915692/1915692]
    -> "data/AGOs-20100831-jdm.tgz" [1]
2015-11-07  15:11:10 URL:http://www.repository.voxforge1.org/downloads/es/
    Trunk/Audio/Main/16kHz_16bit/AGOs-20100831-tmi.tgz [2009016/2009016]
    -> "data/AGOs-20100831-tmi.tgz" [1]
2015-11-07  15:11:15 URL:http://www.repository.voxforge1.org/downloads/es/
    Trunk/Audio/Main/16kHz_16bit/AbdielNavarro-20130409-xvp.tgz
    [2625640/2625640] -> "data/AbdielNavarro-20130409-xvp.tgz" [1]
...
#######  2 - Normalize all the audio files  #######
Normalizing audio files from AGOs-20100831-jdm
Normalizing audio files from AGOs-20100831-tmi
```

Listado A.7 `00-database/run.sh` - Script de generación de la base de datos de VoxForge.

```
#!/usr/bin/env bash

# Create VoxForge Spanish AcousticDataBase

set -e

echo "####### 1 - Download spanish VoxForge audio files #######"
if [ -d data ] ; then
   echo "data folder found, skipping... (to restart, delete the folder)"
else
   ./scripts/getVoxForge.sh
fi

echo "####### 2 - Normalize all the audio files #######"
if [ -d wav ] ; then
   echo "wav folder found, skipping... (to restart, delete the folder)"
else
   ./scripts/normalize.sh
fi

echo "####### 3 - Create transcribed vocabulary from the prompts #######"
./scripts/createTranscVocabulary.sh

echo "####### 4 - Create train.idx/valid.idx partition indexes #######"
./scripts/createTrainValid.sh

echo "####### 5 - Create main.xml AcousticDatabase XML descriptor #######"
./scripts/createAcousticDataBase.sh

echo "####### 6 - Check AcousticDatabase integrity #######"
sautrela AcousticDataBase -cnup "" -r ".*" main.xml
echo "Integrity: OK"
echo "$(sautrela AcousticDataBase -up "" -r "train_.*" main.xml | wc -l) train
    audio resources"
echo "$(sautrela AcousticDataBase -up "" -r "valid_.*" main.xml | wc -l)
    validation audio resources"
```

```
Normalizing audio files from AbdielNavarro-20130409-xvp
...
#######  3 - Create transcribed vocabulary from the prompts  #######
Extracting promts sentences
Transcribing the vocabulary
#######  4 - Create train.idx/valid.idx partition indexes  #######
#######  5 - Create main.xml AcousticDatabase XML descriptor  #######
#######  6 - Check AcousticDatabase integrity  #######
Integrity: OK
21621 train audio resources
1254 validation audio resources
```

A.6. Parametrización

Se denomina parametrización a la la fase de extracción de características relevantes de la señal de audio, es decir, aquellas características que aportan mayor información de discriminación entre las unidades acústicas (fonemas) que serán modeladas en la fase posterior. Dicha información está relacionada con la forma en la que son articulados los sonidos, razón por la cual muchas técnicas de parametrización están relacionadas con la estimación de la envolvente espectral de la señal.

Dado que el proceso de entrenamiento de los modelos acústicos conllevará un gran número de iteraciones sobre un mismo conjunto de señales, es aconsejable realizar primero la parametrización de la base de datos acústica, para reutilizar después, una y otra vez, los recursos ya parametrizados.

En el entorno de desarrollo de Sautrela, la parametrización es configurada mediante el encadenamiento de múltiples fases de procesamiento que son representadas mediante una *Engine* (véase Sección 3.4). El Listado A.8 muestra la representación XML de una engine de parametrización que obtiene los denominados *coeficientes cepstrales en las frecuencias de Mel* (MFCC, por sus siglas en inglés), probablemente los parámetros más comunes en RAH. Cada fase de procesamiento es realizada por un `Processor` (véase Sección 3.2) que es configurable mediante sus correspondientes parámetros (en el Apéndice C puede encontrarse una descripción más detallada de cada procesador y sus correspondientes parámetros). Los procesadores son conectados entre sí mediante `Buffer`s (véase Sección 3.3). La engine de parametrización consta de las siguientes fases de procesamiento:

Lectura de la base de datos Toma el descriptor XML de la base de datos acústica (`main.xml`) y extrae todo su contenido (la señal de audio y las propiedades asociadas a cada recurso).

Preémfasis Aplica un filtro pasa-alto que compensa la atenuación sufrida por las altas frecuencias durante la adquisición de audio.

Ventaneo Toma *instantáneas* temporales de la señal de audio (ventanas temporales de 25ms de longitud), a intervalos de una centésima de segundo. Durante dichas instantáneas, los componentes espectrales pueden considerarse prácticamente constantes.

Espectro de potencia Se obtiene una representación espectral de cada ventana temporal.

Banco de filtros en escala Mel Se obtiene una representación espectral suavizada, usando una escala frecuencial inspirada en la percepción humana.

Transformada discreta de cosenos Obtiene el *cepstrum*, una representación directamente relacionada con la envolvente espectral.

Normalización cepstral Trata de eliminar la componente relativa al canal de transmisión (o cualquier otro efecto convolutivo constante presente en la señal).

Coeficientes_dinámicos Obtiene los coeficientes dinámicos de primer y segundo orden de los cepstrales, para que los parámetros contengan información relativa a la dinámica de la envolvente espectral.

Análisis de la secuencia Se comprueba la integridad de las secuencias de datos.

Volcado Todos los datos (los vectores de cepstrales y las propiedades asociadas a cada recurso) son volcados a un fichero.

Una vez creada la engine, su ejecución es sencilla:

```
$ sautrela ASRParam16k.eng -d ../00-database/main.xml -s "train_.*" -o
    train.dump
$ sautrela ASRParam16k.eng -d ../00-database/main.xml -s "valid_.*" -o
    valid.dump
```

Los ficheros generados, `train.dump` y `valid.dump`, son ficheros binarios de Java (flujos de objetos serializados):

```
$ file train.dump valid.dump
train.dump: Java serialization data, version 5
valid.dump: Java serialization data, version 5
```

Las engines pueden establecer el valor de cualquier parámetro de un procesador, asignándoselo directamente dentro del descriptor XML:

```
<param name="theParamName" value="theValue"/>
```

Sautrela ofrece también la posibilidad de parametrizar una engine, de tal forma que los valores de algunos de sus parámetros puedan ser establecidos en el momento de la ejecución:

```
<param name="theParamName" value="?-optName"/>
```

De esta manera, el parámetro `theParamName` pasa a ser un argumento de la engine (suministrado mediante la opción de comando `-optName`), y su valor por defecto será el establecido por el procesador correspondiente. También es posible fijar un valor por defecto diferente del establecido por el procesador:

```
<param name="theParamName" value="?-optName [newDefaultValue]"/>
```

La parametrización de engines ofrece la posibilidad de crear engines flexibles que puedan adaptarse a diferentes casos de uso, determinando qué parámetros serán prefijados y cuáles serán configurables. La engine del Listado A.8, por ejemplo, establece únicamente tres parámetros configurables: la base de datos, el subconjunto a procesar y el fichero de volcado (cuyo nuevo valor por defecto es `param.dump`).

Dada una engine, es posible consultar la documentación asociada a ella mediante la opción `-help`:

Listado A.8 `01-param/engines/ASRParam16k.eng` - Descriptor XML de una engine de parametrización para el reconocimiento automático del habla. Los parámetros cuyo valor viene dado mediante la expresión "`?-optName`" pasan a ser argumentos (parámetros configurables) de la engine.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE Engine SYSTEM "http://sautrela.org/lib/Engine.dtd">
<Engine name="ASRParam16k"
   description="16kHz audio parametrization for Speech Recognition:&#10;
   &#10;
   13MFCC + Delta-DeltaDelta + CMN&#10;
   &#10;
    - Pre-emphasis&#10;
    - 25ms windows, 10ms shift, HAMMING&#10;
    - PowerSpectrum (FFTsize=512)&#10;
    - MelLogFilterBank (40 filters, minFreq=130.0Hz, maxFreq=6800.0Hz)&#10;
    - DCT (energyIncluded, filters=13)&#10;
    - Deltas&#10;
    - Mean and Variance Normalization (CMVN)">
   <Processor name="ADBReader">
      <param name="databaseURL" value="?-d"/>
      <param name="resourceNameRegex" value="?-s"/>
   </Processor><Buffer size="10"/>
   <Processor name="Preemphasis"/><Buffer/>
   <Processor name="Windowing"/><Buffer/>
   <Processor name="PowerSpectrum"/><Buffer/>
   <Processor name="MelLogFilterBank"/><Buffer/>
   <Processor name="DiscreteCosineTransform"/><Buffer/>
   <Processor name="Delta"/><Buffer/>
   <Processor name="MeanVarianceNormalization"/><Buffer/>
   <Processor name="StreamTester"/><Buffer/>
   <Processor name="StreamWriter" >
      <param name="streamFile" value="?-o [param.dump]"/>
   </Processor>
</Engine>
```

```
$ sautrela -help ASRParam16k.eng

ENGINE: ASRParam16k

  16kHz audio parametrization for Speech Recognition:

  13MFCC + Delta-DeltaDelta + CMN

   - Pre-emphasis
   - 25ms windows, 10ms shift, HAMMING
   - PowerSpectrum (FFTsize=512)
   - MelLogFilterBank (40 filters, minFreq=130.0Hz, maxFreq=6800.0Hz)
   - DCT (energyIncluded, filters=13)
   - Deltas
   - Mean and Variance Normalization (CMVN)
```

```
PARAMETERS:

  -d [URL,file:OPENDIALOG] the locator of the AcousticDataBase or "file:
    OPENDIALOG" for a File Open Dialog [databaseURL,ADBReader]
  -s [String,".*"] the sentences whose names match this regex are read [
    resourceNameRegex,ADBReader]
  -o [File,param.dump] the pathname of the dump file [streamFile,
    StreamWriter]
```

Esta documentación proviene, tanto del descriptor XML de la engine (la documentación descriptiva de la propia engine), como de los procesadores que la conforman (la documentación relativa a los argumentos de la engine). Solo son documentados aquellos parámetros que forman parte del conjunto de argumentos de la engine. Al final de la documentación de cada parámetro puede observarse la procedencia del mismo: el nombre original del parámetro y el procesador al que pertenece. También es posible consultar directamente la documentación de un procesador:

```
$ sautrela -help ADBReader

PROCESSOR: ADBReader (edu.gtts.sautrela.db.ADBReader)

  Reads a subset of an AcousticDataBase

PARAMETERS:

  databaseURL [URL,file:OPENDIALOG] - the locator of the AcousticDataBase
    or "file:OPENDIALOG" for a File Open Dialog
  indexURL [URL,null] - the locator of a Resource Name index resource or "
    file:OPENDIALOG" for a File Open Dialog
  resourceNameRegex [String,".*"] - the sentences whose names match this
    regex are read
```

Por último, conviene recordar que, aunque es posible utilizar señales de audio con características diferentes, los parámetros por defecto de los distintos procesadores de Sautrela han sido establecidos considerando la utilización de señales de audio muestreadas a 16000 hercios. En aquellas situaciones en las que la señal original no cumpla estas condiciones (tal es el caso de las señales telefónicas, cuyo contenido espectral está aproximadamente limitado a la banda de 300-3400 hercios), deberá prestarse especial atención a cada uno de los parámetros de los procesadores.

A.7. Estimación de modelos acústicos

Los modelos acústicos son, sin duda, los componentes más críticos de los sistemas de RAH. Existe una gran variedad de tipos de modelos que pueden ser utilizados para representar una unidad acústica: HMMs discretos o continuos (distribuciones basadas en mezclas de Gaussianas), redes neuronales, o incluso modelos híbridos en los que las distribuciones de los HMMs vienen representadas por redes neuronales. También

existen multitud de criterios de entrenamiento de modelos: máxima verosimilitud (ML por sus siglas en inglés), máximo a posteriori (MAP por sus siglas en inglés), máxima información mutua (MMI por sus siglas en inglés), etc. Por último, el objeto a modelar (lo que llamamos unidad acústica) puede definirse también de muchas formas: es posible modelar las unidades fonéticas de manera incontextual o definir múltiples unidades acústicas en base a las posibles alternativas de considerar contextos a izquierda o derecha (unidades contextuales), etc.

El presente apartado muestra un sencillo ejemplo de estimación por ML de modelos acústicos incontextuales (comúnmente denominados *monofonemas*) en los que cada unidad fonética es representada mediante un HMM de topología secuencial con bucles, compuesto por tres estados cuyas distribuciones de probabilidad vienen dadas por mezclas de Gaussianas.

El Listado A.9 muestra un script que lleva a cabo la estimación de dichos modelos. Siguiendo un esquema estándar para la estimación de HMMs continuos, se parte de una única Gaussiana que es estimada sobre el conjunto completo de muestras de entrenamiento. Dicha Gaussiana sirve de semilla para la confección de los modelos acústicos iniciales que contarán con una única componente Gaussiana. Los modelos acústicos representan a cada uno de los fonemas del inventario fonético, más el modelo correspondiente al silencio. Dichos modelos son re-estimados a partir de los recursos de audio del subconjunto de entrenamiento y la transcripción fonética creada previamente. Dado que las transcripciones no contemplan silencios, durante el entrenamiento se permite la inserción libre de silencios en cualquier lugar de la transcripción[8]. Una vez realizadas diez iteraciones de re-estimación, se lanzan sendos experimentos de decodificación acústico-fonética sobre los subconjuntos de entrenamiento y validación, para monitorizar el proceso de re-estimación. El número de componentes de cada modelo es duplicado y el proceso de re-estimación es reiniciado una y otra vez, hasta obtener el tamaño o número de componentes Gaussianas deseado (16 en el caso que nos atañe).

El script del Listado A.9 hace uso de tres engines adicionales: la Engine `GMMTrainer.eng` (véase Listado A.10) diseñada para entrenar un GMM a partir de un volcado de datos, la Engine `WFSATrainer.eng` (véase Listado A.11) que permite entrenar un conjunto de modelos acústicos a partir de un volcado de datos que contenga transcripciones y por último la Engine `DecodeAndRates.eng` (véase Listado A.12), que realiza la decodificación acústico-fonética, obtiene la tasa de reconocimiento (el valor del índice de precisión) y vuelca las secuencias reconocidas a un fichero de datos. La Figura A.2 muestra la evolución de la tasa de reconocimiento para los subconjuntos de entrenamiento y validación a lo largo del proceso de re-estimación de los modelos[9].

[8] Para facilitar la convergencia de los modelos, las primeras cinco iteraciones de entrenamiento solo permiten la inserción de silencios al inicio y final del recurso de audio (opción `-tim lr` de la Engine `WFSATrainer.eng`), mientras que a partir de la sexta iteración la inserción de silencios puede darse en cualquier posición (opción `-tim lir` de la Engine `WFSATrainer.eng`, su valor por defecto) .

[9] Para agilizar el proceso de re-estimación, el script del Listado A.9 únicamente obtiene las tasas tras la última iteración de re-estimación de modelos para cada uno de los tamaños utilizados (1, 2, 4, 8 y 16 componentes). No obstante, unos pequeños cambios en dicho script bastan para obtener el conjunto completo de valores de la Figura A.2.

Listado A.9 `02-acoustic/run.sh` - Script de estimación de modelos acústicos.

```
#!/usr/bin/env bash

# Train monophone acoustic models

source scripts/config
# AcousticDataBase
DB="../00-database/main.xml"
# train/valid parameterized data files
TRAIN="../01-param/train.dump"
VALID="../01-param/valid.dump"
# Beam search value
BEAM=50
# Feature dimenssion
DIM=39
# Number of Gaussian components
GAUSS=16
# Number of iterations (total number of iteration is (1+log2(GAUSS))*ITER)
ITER=10


echo "####### Get the phone inventory #######"
sautrela AcousticDataBase -nup "Phon" -r ".*" $DB \
  | awk '$1=$NF' | tr " " "\n" | sort | uniq > phonelist
PHONES="$(tr "\n" "," < phonelist)SIL"
rm phonelist && echo $PHONES

echo "####### Train a single component GMM #######"
sautrela GMM -SMVW -d $DIM > init.gmm
sautrela engines/GMMTrainer.eng -d $TRAIN -i init.gmm -o init.gmm

echo "####### Create initial HMMs #######"
sautrela CHMM -n initial -g init.gmm 3 > init.hmm
sautrela WFSASet init.hmm > init.hmmset
sautrela CHMMSetEdit -n $PHONES init.hmmset > phones.1g.wfsaset
rm init.gmm init.hmm init.hmmset

echo "####### Train acoustic models from transcriptions #######"
mkdir rec
for((g=1;g<=GAUSS;g*=2)); do
   m=phones.${g}g.wfsaset
   if ((g > 1)) ; then sautrela CHMMSetEdit -g $g $mold > $m ;fi
   echo "####### Training $m #######"
   for((i=1;i<=ITER;i++)); do
      echo "####### Training $m ($i/$ITER) $(date) #######"
      if ((g==1 && i <= 5)) ; then
         sautrela engines/Trainer.eng -d $TRAIN -beam $BEAM -tim lr -i $m -o $m
      else
         sautrela engines/Trainer.eng -d $TRAIN -beam $BEAM -i $m -o $m
      fi
   done
   sautrela LMM -e $m > PhoneDecoder.lmm
   sautrela engines/DecodeAndRates.eng -dIn $TRAIN -dOut rec/train.${g}g.dump \
     -beam $BEAM -i PhoneDecoder.lmm > rec/train.${g}g.out 2> rec/train.${g}g.err &
   sautrela engines/DecodeAndRates.eng -dIn $VALID -dOut rec/valid.${g}g.dump \
     -beam $BEAM -i PhoneDecoder.lmm > rec/valid.${g}g.out 2> rec/valid.${g}g.err &
   mold=$m
done
waitAll
```

Listado A.10 `02-acoustic/engines/GMMTrainer.eng` - Descriptor XML de una engine genérica para el entrenamiento de un GMM. El primer procesador es el encargado de leer las secuencias de datos desde un fichero de volcado mientras que el segundo se encarga de la re-estimación del GMM suministrado.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE Engine SYSTEM "http://sautrela.org/lib/Engine.dtd">
<Engine name="GMMTrainer" description="Trains a GMM from a dumped data file">
  <Processor name="StreamReader">
    <param name="streamURL" value="?-d [data.dump]" />
  </Processor><Buffer/>
  <Processor name="GMMTrainer">
    <param name="modelURL" value="?-i [in.gmm]" />
    <param name="outputFile" value="?-o [out.gmm]" />
    <param name="threadNumber" value="?-t" />
   <param name="chunkSize" value="?-c" />
   <param name="mapCount" value="?-m" />
    <param name="topN" value="?-n" />
  </Processor>
</Engine>
```

Listado A.11 `02-acoustic/engines/WFSATrainer.eng` - Descriptor XML de una engine de entrenamiento de WFSAs. El primer procesador es el encargado de leer las secuencias de datos desde un fichero de volcado y el segundo es el que realiza la re-estimación de los modelos.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE Engine SYSTEM "http://sautrela.org/lib/Engine.dtd">
<Engine name="WFSA/WFSASets Trainer" description="Trains WFSA/WFSASets from a
     dumped data file.">
  <Processor name="StreamReader">
    <param name="streamURL" value="?-d" />
  </Processor><Buffer/>
    <Processor name="Trainer">
    <param name="modelURLList" value="?-i" />
    <param name="outputFileList" value="?-o" />
    <param name="beam" value="?-beam" />
    <param name="transcInsertionMask" value="?-tim [lir]" />
    <param name="transcInsertionSymbol" value="?-tis [SIL]" />
    <param name="transcPropertyName" value="?-tpn [Phon]" />
    <param name="gcPolicy" value="?-gcp [GC]" />
    <param name="gcMinInterval" value="?-gci [4000]" />
    <param name="verbose" value="?-v" />
  </Processor>
</Engine>
```

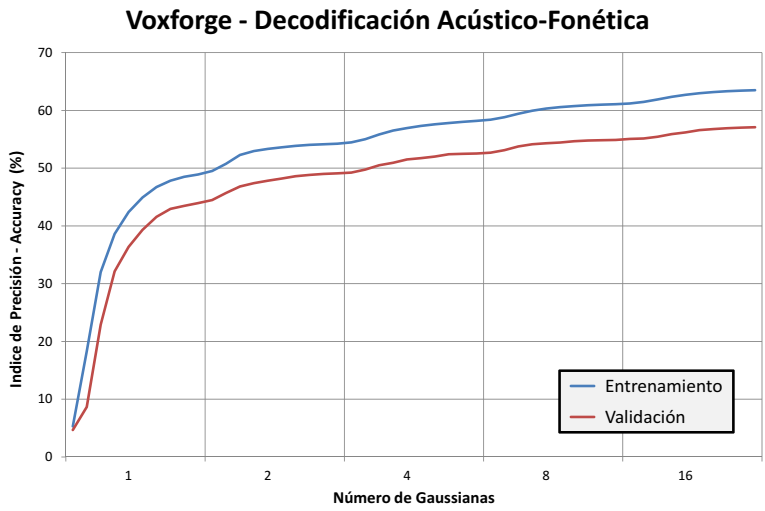


Figura A.2 Evolución del índice de precisión fonético durante la re-estimación de los modelos acústicos.

Listado A.12 `02-acoustic/engines/DecodeAndRates.eng` - Descriptor XML de una engine de decodificación y estimación de tasas. El primer procesador es el encargado de leer las secuencias de datos desde un fichero de volcado, el segundo obtiene la decodificación más probable en base al WFSA suministrado, el tercero obtiene la tasa de reconocimiento y el último de ellos vuelca el resultado de la decodificación a otro fichero de datos. El fichero volcado sirve para la realización de posteriores análisis del proceso de decodificación.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE Engine SYSTEM "http://sautrela.org/lib/Engine.dtd">
<Engine name="WFSASet Decoder with RecognitionRate">
  <Processor name="StreamReader">
    <param name="streamURL" value="?-dIn" />
  </Processor><Buffer/>
  <Processor name="Decoder">
    <param name="beam" value="?-beam" />
    <param name="modelURL" value="?-i" />
   <param name="gcPolicy" value="?-gcp [GC]" />
    <param name="gcMinInterval" value="?-gci [4000]" />
    <param name="verbose" value="?-v" />
  </Processor><Buffer/>
  <Processor name="RecognitionRate">
    <param name="rate" value="?-r" />
   <param name="transcPropertyName" value="?-t [Phon]" />
    <param name="excludeSymbolPattern" value="?-e [SIL]" />
  </Processor><Buffer/>
  <Processor name="StreamWriter">
    <param name="streamFile" value="?-dOut" />
  </Processor>
</Engine>
```

A.8. Estimación mejorada mediante filtrado de envíos

Como ya se ha comentado previamente, las grabaciones descargadas pueden contener audio ruidoso, locuciones que no coincidan con la transcripción o, incluso, ningún audio. El módulo de entrenamiento de Sautrela es, en cierta manera, robusto ante este hecho, ya que descartará todas aquellas señales cuyo contenido diste demasiado de su transcripción. Sin embargo, todas aquellas señales ruidosas o con un número aún considerable de errores de transcripción o pronunciación formarán parte del conjunto de entrenamiento. Nótese que el uso de señales ruidosas o con transcripciones defectuosas puede dar lugar a modelos acústicos de bajo rendimiento si dichos factores no son debidamente tenidos en cuenta o compensados.

Ante esta circunstancia, una posible alternativa es partir del conjunto completo de grabaciones, realizar el proceso completo de entrenamiento de modelos acústicos del apartado anterior y, a partir de los modelos estimados, descartar los envíos[10] de menor verosimilitud. La Figura A.3 muestra el ranking de los envíos del subconjunto de entrenamiento según la precisión fonética calculada a partir de un conjunto de modelos estimado con el conjunto completo de entrenamiento descargado de VoxForge. Cabe esperar que la probabilidad de error en los envíos sea inversamente proporcional a su precisión fonética, por lo que los envíos con menor precisión deberían ser descartados a la hora de entrenar modelos acústicos. La Tabla A.1 muestra en su primera fila el rendimiento acústico del sistema entrenado con el conjunto completo de datos. Cada una de las filas inferiores muestra el rendimiento de un sistema entrenado únicamente a partir de los n mejores envíos. Nótese que al reducir el tamaño de la base de datos, reducimos también el número de locutores[11] y el tiempo requerido para estimar los modelos acústicos. En este caso concreto, la exclusión de los envíos de menor precisión no ofrece una mejoría significativa en los resultados finales. Sin embargo, sí que conlleva un ahorro considerable de tiempo en el proceso de estimación de los modelos, todo ello sin sacrificar la precisión obtenida para el conjunto de validación.

El Listado A.13 contiene un script que, en vez de descargar el conjunto completo de envíos (2173), descarga únicamente los n envíos de mayor precisión acústica. Este script hace uso de los ficheros índice `voxforge.train.16g.Accuracy.sorted` y `voxforge.valid` que contienen el índice ordenado de envíos de entrenamiento y el índice de envíos de validación respectivamente. Ambos ficheros pueden encontrarse junto al resto de ficheros del presente Apéndice. El siguiente apartado muestra el proceso de obtención del índice ordenado de envíos de entrenamiento.

Obtención del ranking de envíos

La sección anterior ha mostrado el proceso de estimación de modelos acústicos. Producto de dicho proceso, se generará un directorio `rec` que contiene el resultado de los procesos de decodificación: las tasas de precisión fonética y el volcado de la decodificación:

[10] La mayoría de los envíos consta de 10 grabaciones.

[11] A la hora de calcular el número de locutores, todos los envíos anónimos son considerados como de un mismo locutor, por lo que cabe esperar que el número real de locutores sea mayor.

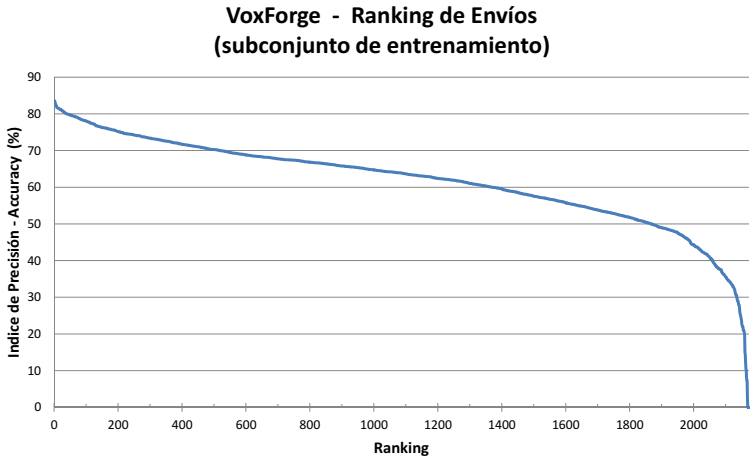


Figura A.3 Ranking de precisión fonética de los envíos del subconjunto de entrenamiento de VoxForge. Una tasa demasiado baja indica indudablemente la existencia de errores en el envío.

Listado A.13 `00-database/scripts/getVoxForgeBest.sh` - Script de descarga de lo n envíos de mayor precisión acústica de VoxForge.

```
#!/usr/bin/env bash

# Download the "best" recordings from the spanish voxforge database (16kHz,16bit)
# NOTE: best --> the goodness of the recordings is measured as the accuracy based
# on a 16 Gaussians decoding system trained on the full dataset

if (( $# != 1 )) ; then
   echo " Syntax: $0 number_of_train_submissions" > /dev/stderr
   exit 1
fi
source scripts/config
URL="http://repository.voxforge1.org/downloads/es/Trunk/Audio/Main/16kHz_16bit"
TRAINLIST="scripts/voxforge.train.16g.Accuracy.sorted"
VALIDLIST="scripts/voxforge.valid"


for x in $(head -$1 $TRAINLIST | awk 'NF=1') $(cat $VALIDLIST) ; do
   echo $URL/$x.tgz
done > wget.list

mkdir data
wget --no-verbose -P data -i wget.list
for f in data/*.tgz ; do
   echo $f
   tar -xzC data -f $f
   rm $f
done
```

n	Tamaño de Entrenamie. (horas)	Locutores	Precisión (accuracy) %		Proceso de Entrenamie. (hh:mm:ss)
			Entrenamie.	Validación	
2173	51,5	460	63.5	57.1	20:11:03
2000	45.7	400	66.2	57.6	16:33:32
1800	41.1	352	67.9	57.5	15:13:56
1600	36.5	319	69.3	57.1	13:27:15
1400	32.0	275	70.7	56.9	11:24:21
1200	27.3	228	72.1	57.0	9:27:14
1000	22.6	174	73.7	57.3	8:05:42
800	18.0	149	75.3	57.7	6:14:49
600	13.5	117	77.4	57.5	4:38:52
400	9.0	73	80.2	56.7	3:13:03
200	4.5	31	85.1	55.5	1:30:31

Tabla A.1 Rendimiento (precisión fonética) de los subconjuntos optimizados de VoxForge en función del número n de envíos seleccionados. La selección se basa en los n envíos de mayor precisión para el conjunto de validación, usando modelos acústicos de 16 componentes Gaussianas entrenados sobre el conjunto completo de datos (primera fila).

```
$ cat rec/train.16g.out
Rate (ACC - Accuracy):  61.720055475563306
Corrects:       1161945
Substitutions:  384369
Deleted:        114958
Inserted:       136607
$ cat rec/valid.16g.out
Rate (ACC - Accuracy):  56.45457466831934
Corrects:       30026
Substitutions:  10784
Deleted:        2756
Inserted:       5431
$ file rec/train.16g.dump rec/valid.16g.dump
rec/train.16g.dump: Java serialization data, version 5
rec/valid.16g.dump: Java serialization data, version 5
```

La engine del Listado A.12 es la utilizada para la decodificación, la obtención de tasas y el volcado final. El flujo de salida del decodificador no es alterado por el Procesador `RecognitionRate`, por lo que los ficheros volcados contienen el flujo de salida original de los decodificadores. Dichos ficheros están compuestos por secuencias de cadenas de caracteres, cada una de ellas con su correspondiente cabecera, donde se encuentra la transcripción original. La precisión fonética de un envío puede ser obte-

nida cargando únicamente las secuencias pertenecientes a dicho envío y volviéndolas a procesar con el Procesador `RecognitionRate`. El Listado A.14 muestra una engine diseñada para la obtención selectiva de tasas a partir de la salida volcada de un decodificador. Las secuencias cargadas son seleccionadas usando como criterio alguna de las propiedades de la cabecera: son seleccionadas únicamente aquellas secuencias cuyas cabeceras contengan la propiedad indicada y su valor sea representado por el patrón (expresión regular) suministrado. Es sencillo obtener las tasas del conjunto completo, ya que por defecto serán seleccionadas todas las secuencias:

```
$ sautrela engines/FilteredRates.eng -i rec/train.16g.dump
Rate (ACC - Accuracy):  61.720055475563306
Corrects:       1161945
Substitutions:  384369
Deleted:        114958
Inserted:       136607
```

Para obtener la tasa de un envío específico, precisamos de alguna propiedad que pueda identificarlo. En el caso de nuestra base de datos, el localizador de un recurso (propiedad `ResourceURL`) cumple dicha función, ya que los localizadores de los recursos de entrenamiento de la base de datos previamente diseñada tienen el esquema:

```
file:../00-database/wav/ID_ENVIO/es-00XX.wav
```

donde `ID_ENVIO` se refiere justamente al identificador de envío. Por ejemplo, la precisión del envío `AGOs-20100831-jdm` puede obtenerse mediante:

```
$ sautrela engines/FilteredRates.eng -d rec/train.16g.dump \
    -pName "ResourceURL" -pPattern ".*/AGOs-20100831-jdm/.*"
Rate (ACC - Accuracy):  72.87630402384501
Corrects:       538
Substitutions:  123
Deleted:        10
Inserted:       49
```

El Listado A.15 muestra un script que obtiene el ranking de los envíos del subconjunto de entrenamiento y genera los ficheros `voxforge.train.16g.Accuracy.sorted` y `voxforge.valid`.

A.9. Modelos léxicos

Una vez entrenados los modelos acústicos, el siguiente paso consiste en la creación de los modelos léxicos que representen las palabras del vocabulario a reconocer. El conjunto de comandos de Sautrela permite crear un conjunto de modelos léxicos a partir de un fichero de texto que contenga, en cada línea, una palabra y su correspondiente transcripción fonética: primero, se crea un diccionario (usando el formato XML de Sautrela, que evita problemas de codificación), y posteriormente se crea un conjunto de modelos deterministas a partir del diccionario:

Listado A.14 `02-acoustic/engines/FilteredRates.eng` - Descriptor XML de una engine para la obtención selectiva de tasas a partir de la salida de un decodificador. Las secuencias cargadas son filtradas de acuerdo a a alguna de las propiedades de la cabecera.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE Engine SYSTEM "http://sautrela.org/lib/Engine.dtd">
<Engine name="Filtered RecognitionRate from dumped decodings">
  <Processor name="StreamReader">
    <param name="streamURL" value="?-d" />
  </Processor><Buffer/>
  <Processor name="StreamGrep">
    <param name="propertyName" value="?-pName" />
    <param name="propertyValueRegex" value="?-pPattern" />
  </Processor><Buffer/>
  <Processor name="RecognitionRate">
    <param name="transcPropertyName" value="?-t [Phon]" />
    <param name="excludeSymbolPattern" value="?-e [SIL]" />
    <param name="rate" value="?-r" />
  </Processor>
</Engine>
```

Listado A.15 `00-database/scripts/rankVoxForge.sh` - Script de obtención del ranking de entrenamiento de VoxForge.

```
#!/usr/bin/env bash

# Rank VoxForge train submissions by phonetic accuracy

source scripts/config
saveFunction onExit _onExit
function onExit { _onExit ; rm -r rank ; }
DB="../00-database/main.xml"
DUMP="rec/train.16g.dump"
# Number of parallel processes
NUMPROCS=8
# get VoxForge submission IDs by resource's name pattern
function getIDs {
   sautrela AcousticDataBase -np "" -r "$1" $DB \
     | awk -F / '{print $(NF-1)}' | sort | uniq
}
# correct sort with "." decimal separator
export LC_ALL=C


mkdir rank
getIDs "train_.*" | while read id ; do
   echo sautrela engines/FilteredRates.eng -d $DUMP -pName ResourceURL \
     -pPattern ".*/$id/.*" " -t Phon -e SIL > rank/$id.acc"
done | ./scripts/parRun.sh $NUMPROCS
for f in $(ls rank) ; do
   echo ${f%.acc} $(awk 'NR==1{print $NF}' rank/$f)
done | sort -grk2 > voxforge.train.16g.Accuracy.sorted

getIDs "valid_.*" > voxforge.valid
```

```
$ head -5 voc.transc
acústico a k u s t i k o
adjetivo a d x e t i b o
aliño a l i h o
almacén a l m a z e n
susurró s u s u @ o
$ sautrela Dictionary voc.transc > dictionary
$ sautrela DefaultDWFSASet dictionary > lexicon.wfsaset
```

Nuestro objetivo será la construcción de un sistema de RAH que será evaluado sobre el conjunto de validación de la base de datos VoxForge. Dicho conjunto consta de 140 frases repetidas por 9 locutores distintos, y las frases carecen de un contexto común: el conjunto de frases no pertenecen a una tarea o tema concreto. Debido a esta limitación, el sistema de RAH que crearemos a continuación hará uso expreso del conjunto de frases de evaluación, extrayendo de allí toda la información relativa al léxico y al lenguaje, dejando el estudio de un sistema de RAH más realista para la última sección.

El Listado A.16 muestra un script que extrae el conjunto de frases de validación, obtiene de ellas el vocabulario, lo transcribe y crea un conjunto de modelos léxicos listos para ser integrados en un sistema de RAH. Además, dichos modelos léxicos podrán contener opcionalmente un fonema de silencio inicial o final[12]. El conjunto de modelos resultante está compuesto por 580 unidades léxicas, un tamaño de vocabulario que podríamos considerar *pequeño* si lo comparamos con los vocabularios que manejan los sistemas de dictado tradicionales.

A.10. Modelo de lenguaje

De manera análoga a lo dispuesto en la sección anterior, podemos entrenar un modelo de lenguaje a partir del conjunto de frases de validación. Debemos tener muy en cuenta que al entrenar un modelo de lenguaje sobre el propio conjunto de frases que vamos a reconocer, el rendimiento del sistema estará gravemente sesgado. No obstante, este ejercicio servirá para mostrar cuáles son los pasos a seguir para la creación y estimación de un modelo de lenguaje a partir de texto.

El Listado A.17 muestra un script que crea y entrena tres modelos de lenguaje basados en n-gramas (1-gramas, 2-gramas y 3-gramas, respectivamente). Los tres modelos son estimados a partir del conjunto de frases obtenido en la sección anterior. El entrenamiento es llevado a cabo por la engine del Listado A.18, que carga un recurso

[12]Los silencios pueden ser tratados a nivel fonético (como unidades fonéticas), a nivel léxico (como unidades léxicas) o a ambos niveles. Sin embargo, debe tenerse en cuenta que si se define una unidad léxica que represente un silencio, posiblemente se degradará la información proveniente del modelo de lenguaje, que probablemente no los habrá considerado en su estimación. Existen dos alternativas sencillas: a) permitir unidades de silencio en los bordes de las palabras, o b) permitir unidades de silencio únicamente en uno de los extremos de las palabras y añadir una unidad léxica de silencio que tenderá a aparecer solo en uno de los extremos de las frases reconocidas (y que, por tanto, no degradará la información proveniente del modelo de lenguaje).

Listado A.16 `03-lexicon/run.sh` - Script de creación de los modelos léxicos de VoxForge.

```
#!/usr/bin/env bash

# Create VoxForge validation set lexical models

source scripts/config
# AcousticDataBase
DB="../00-database/main.xml"
# get phonetic transcription from UTF-8 encoded prompts
function ortho2phon {
   iconv -f UTF-8 -t LATIN1 \
     | ../00-database/scripts/ehu_transcribe.LATIN1.pl -w \
     | iconv -f LATIN1 -t UTF-8
}
# split all the character: "inputtext" --> " i n p u t t e x t"
function splitWord { sed -e "s/./ &/g" ; }


echo "Extracting the VoxForge validation sentences"
sautrela AcousticDataBase -nup Ortho -r "valid.*" $DB | cut -d " " -f 3- \
  | sort | uniq > sentences.txt

echo "Extracting the vocabulary"
cat sentences.txt | tr " " "\12" | sort | uniq > voc

echo "Transcribing vocabulary"
cat voc | ortho2phon | splitWord | paste -d "" voc - > voc.transc

echo "Creating a Dictionary"
sautrela Dictionary voc.transc > dictionary

echo "Creating the lexicon (i.e. the WFSASet containing lexical models)"
sautrela DefaultDWFSASet -lr -s SIL dictionary > lexicon.wfsaset
```

de texto, genera secuencias textuales a partir de las líneas de texto y re-estima los parámetros del WFSA suministrado.

A.11. Modelo Integrado

Una vez estimados los conjuntos de modelos que representan cada capa de conocimiento (modelos acústicos, léxicos y de lenguaje), el siguiente paso es preparar dichos modelos para que puedan ser utilizados en un proceso de decodificación. El decodificador de Sautrela procesa (decodifica) las secuencias de entrada en base a un único modelo WFSA, por lo que se debe construir un modelo unificado que integre todas las capas de conocimiento dentro de un único WFSA.

Este modelo unificado apareció ya en la Sección A.7, al realizar experimentos de decodificación acústico-fonética para la monitorización del proceso de re-estimación de modelos acústicos. En aquel caso, el script del Listado A.9 hacía uso del comando `LMM` para generar un WFSA que posteriormente era utilizado en la decodificación. El

Listado A.17 `04-language/run.sh` - Script de creación de los modelos de lenguaje de *1*-gramas, *2*-gramas y *3*-gramas para VoxForge. Los tres modelos son entrenados a partir de las frases del conjunto de validación.

```
#!/usr/bin/env bash

# Train 1-gram, 2-gram & 3-gram LM from VoxForge validation sentences

source scripts/config
# Training sentences
SENT="../03-lexicon/sentences.txt"
# Vocabulary
VOC="../03-lexicon/voc"
# Expected vocabulary size (must be higher than the training vocabulary size)
vocSize=$(($(wc -l < $VOC) + 1))


echo "Creating Initial n-gram language models"
sautrela KTLSS -N 1 -v $vocSize > 1gram.ktlss &
sautrela KTLSS -N 2 -v $vocSize > 2gram.ktlss &
sautrela KTLSS -N 3 -v $vocSize > 3gram.ktlss &
waitAll

echo "Training the n-gram language models"
sautrela engines/WFSATxtTrainer.eng -txt $SENT -i 1gram.ktlss -o 1gram.ktlss &
sautrela engines/WFSATxtTrainer.eng -txt $SENT -i 2gram.ktlss -o 2gram.ktlss &
sautrela engines/WFSATxtTrainer.eng -txt $SENT -i 3gram.ktlss -o 3gram.ktlss &
waitAll
```

Listado A.18 `04-language/engines/WFSATxtTrainer.eng` - Descriptor XML de una engine para entrenamiento de WFSAs a partir de texto. La engine carga un recurso de texto y re-estima el WFSA a partir de las secuencias de texto (cada línea genera una secuencia de datos textuales).

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE Engine SYSTEM "http://sautrela.org/lib/Engine.dtd">
<Engine name="txt2WFSA" description="trains a WFSA from text input">
   <Processor name="TextReader">
      <param name="charset" value="?-c"/>
      <param name="textURL" value="?-txt"/>
      <param name="deleteRegex" value="?-d"/>
      <param name="splitRegex" value="?-s"/>
      <param name="toUpperCase" value="?-u"/>
      <param name="toLowerCase" value="?-l"/>
   </Processor><Buffer/>
   <Processor name="Trainer">
      <param name="modelURLList" value="?-i"/>
      <param name="outputFileList" value="?-o"/>
   </Processor>
</Engine>
```

comando `LMM` crea un WFSA denominado modelo de Markov por capas (LMM, por sus siglas en inglés) que integra en un único WFSA no-determinista un número arbitrario de capas de conocimiento. Cada capa es representada por un conjunto de modelos, a excepción de la capa del extremo superior, que es representada por un único modelo:

```
$ sautrela -help LMM

  LMM (edu.gtts.sautrela.wfsa.models.LMM)

    create a Layered Markov Model from WFSASets

  Syntax:  LMM  [-c]  [-d dLayer]  [-n name]  <-e | -s | wfsaURL [a b]>  <
    wfsasetURL [a b]> ...

    -c  Check the consistency of the connected layers
    -d dLayer  The index [0,n-1] (bottom-up)) of the Layer to be used for
    decoding (default to n-1, top-layer)
    -n name  The name of the LMM
    -e  Equiprobable top Layer (accepts any combination of lower layer
    models)
    -s  Selector top Layer (accepts any single model from the lower layer)
    wsfa  The locator of a WFSA to be used as top Layer
    wsfaset  The locator of a WFSASet for the next Layer
    a  Log scale applied to this layer probabilities
    b  Log offset applied to this layer probabilities
```

En el caso de la decodificación acústico-fonética implementada en el script del Listado A.9, el objetivo era obtener, dada una secuencia de vectores de parámetros de entrada, la secuencia fonética más probable, sin contar con conocimiento fonotáctico previo alguno (procesamiento comúnmente denominado lazo-abierto u *open-loop*). Para ello, se ejecutaba el siguiente comando:

```
$ sautrela LMM -e phones.wfsaset > PhoneDecoder.lmm
```

que genera un LMM de dos capas: la capa inferior contiene a los modelos acústicos (`phones.wfsaset`), y la capa superior (opción `-e`) contiene a un único modelo que otorga la misma probabilidad a toda secuencia de fonemas (un modelo de distribuciones equiprobables sobre el conjunto de secuencias fonéticas).

En el caso que nos atañe, debemos integrar las capas representadas por los modelos acústicos, léxicos y de lenguaje en un único LMM. El comando a ejecutar podría ser:

```
$ sautrela LMM language.wfsa lexicon.wfsaset phones.wfsaset > ASR.lmm
```

Sin embargo, un sistema de RAH estándar suele permitir la ponderación del aporte de cada una de las capas de conocimiento al proceso de búsqueda de la secuencia óptima. Normalmente, suele ser posible configurar sendos pesos para los modelos de lenguaje y los modelos acústicos, así como una penalización de inserción de palabras. Estos parámetros pueden ser interpretados, más sencillamente, como una transforma-

ción afín de los logaritmos de las probabilidades de los conjuntos de modelos. Así, un pesado de los modelos de lenguaje o acústicos no es más que un escalado del logaritmo de sus probabilidades, mientras que una penalización de inserción de palabra representa un desplazamiento negativo al logaritmo de las probabilidades del modelo de lenguaje[13].

Los modelos LMM permiten definir una transformación afín sobre los logaritmos de las probabilidades de cada una de las capas. Para ello, es necesario hacer uso de los parámetros `a` y `b` del comando `LMM` (sus valores por defecto son 1 y 0, que equivalen a no transformar las probabilidades):

```
$ sautrela LMM language.wfsa 4 -1 lexicon.wfsaset 2 1 \
    phones.wfsaset 10 3 > ASR.lmm
```

El rendimiento del sistema puede variar drásticamente en función de la elección de dichos parámetros. La Figura A.4 muestra un mapa de calor para los valores del índice de precisión léxica (*word accuracy*) sobre el conjunto de validación de VoxForge para los cuatro sistemas previamente mencionados, realizando únicamente un barrido de los parámetros de la capa de lenguaje. Puede observarse que el valor del índice de precisión léxica puede variar desde alrededor de un 20 % (transformación por defecto) hasta casi un 90 % [14].

A.12. Decodificación

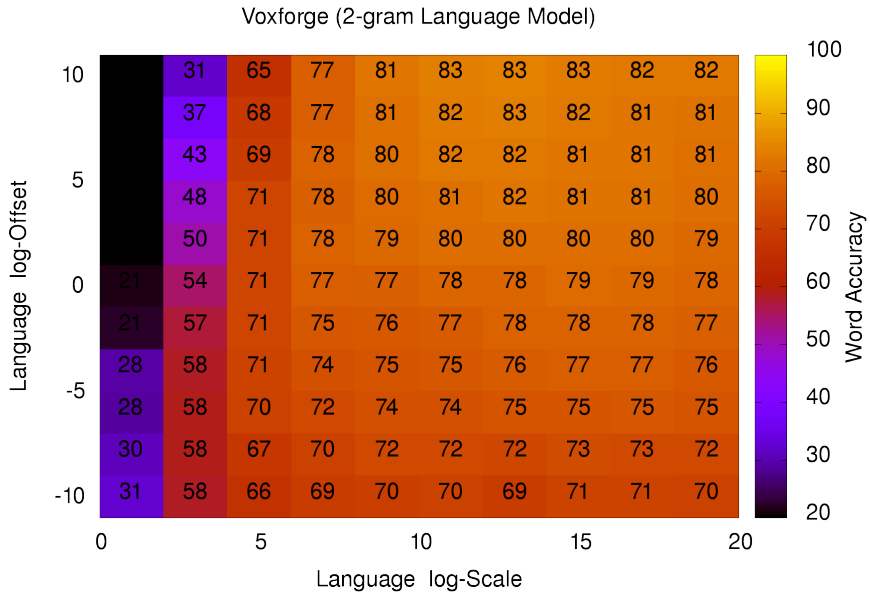
Dado que la decodificación de Sautrela se realiza siempre frente a un único modelo, una sencilla decodificación acústico-fonética es básicamente equivalente a una compleja decodificación acústico-léxica con modelo de lenguaje de n-gramas. Debido a ello, el proceso de decodificación será muy similar al llevado a cabo en la Sección A.7, pudiendo hacer uso de la misma engine de decodificación que obtenía ademas las tasas de reconocimiento. El único factor diferencial reside justamente en el proceso de obtención de las tasas de reconocimiento. En el caso de la decodificación acústico-fonética, las tasas eran obtenidas a partir de la propiedad `Phon` de la base de datos acústica, la cual contenía la transcripción fonética de cada secuencia; en este caso, la salida del decodificador será una secuencia de palabras, por lo que, dada nuestra base de datos acústica, las tasas deberán obtenerse a partir de la propiedad `Ortho`.

El Listado A.19 muestra una engine muy similar a la utilizada en la decodificación acústico-fonética (véase Listado A.12). Las diferencias más significativas son la modificación del valor por defecto del parámetro `transcPropertyName` (de `Phon` a `Ortho`) y el uso del parámetro `averageTrellisSize` que activa el mecanismo de ajuste automático de ancho de haz (*Automatic Beam Tuning*). Este mecanismo hace uso de las primeras secuencias de entrada para ajustar el valor del ancho de haz, de tal manera que el número promedio de estados activos en cada instante sea el deseado. El mecanismo

[13] Cada vez que se evalúa una probabilidad de lenguaje, ello se debe a la posible inserción de una nueva palabra, por lo que un desplazamiento negativo equivale a una penalización de inserción.

[14] Al haber entrenado el modelo de lenguaje sobre el conjunto de frases de validación, la promoción de las probabilidades de lenguaje sobre las probabilidades acústicas mejora el rendimiento del sistema de una manera poco realista.

a)



b)

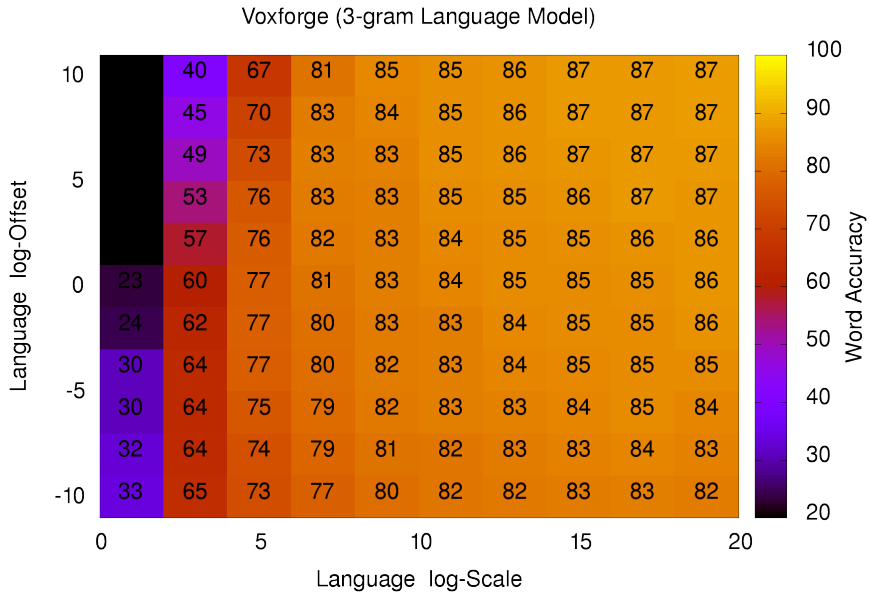


Figura A.4 Mapa de calor para los valores del índice de precisión léxica (word accuracy) sobre el conjunto de validación de VoxForge para un sistema de RAH: a) con modelo de lenguaje de 2-gramas, y b) con modelo de lenguaje de 3-gramas. El mapa de calor muestra los valores del índice de precisión para diferentes configuraciones de los parámetros de escalado y desplazamiento logarítmico de las probabilidades del modelo de lenguaje.

Listado A.19 `05-asr/engines/DecodeAndRates.eng` - Descriptor XML de una engine para la decodificación y obtención de tasas a partir de un volcado de datos.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE Engine SYSTEM "http://sautrela.org/lib/Engine.dtd">
<Engine name="WFSA Decoder and RecognitionRate">
  <Processor name="StreamReader">
    <param name="streamURL" value="?-dIn" />
  </Processor><Buffer/>
  <Processor name="Decoder">
    <param name="beam" value="?-beam" />
    <param name="averageTrellisSize" value="?-ats" />
    <param name="modelURL" value="?-i" />
   <param name="gcPolicy" value="?-gcp [GC]" />
    <param name="gcMinInterval" value="?-gci [0]" />
    <param name="verbose" value="?-v" />
  </Processor><Buffer/>
  <Processor name="RecognitionRate">
    <param name="rate" value="?-r" />
   <param name="transcPropertyName" value="?-t [Ortho]" />
    <param name="excludeSymbolPattern" value="?-e" />
  </Processor><Buffer/>
  <Processor name="StreamWriter">
    <param name="streamFile" value="?-dOut" />
  </Processor>
</Engine>
```

de ajuste automático es especialmente útil cuando se aplican transformaciones afines a las distribuciones de probabilidad de los modelos, ya que ello modifica seriamente su rango dinámico (un mismo ancho de haz puede resultar excesivamente grande en unos casos y excesivamente pequeño en otros).

El Listado A.20 muestra un script que, tras generar dos modelos integrados en base a las diferentes configuraciones de modelos de lenguaje, evalúa su rendimiento sobre el conjunto de validación de Voxforge. Los valores de las transformaciones de las distribuciones de probabilidad han sido optimizados sobre el propio conjunto de validación, y los valores del índice de precisión obtenidos son de 84,5 % y 90,9 %, para los sistemas basados en 2-gramas y 3-gramas, respectivamente:

```
$ ./run.sh
Building the LMM models
Running the decoders
2gram Language Model - Word Accuracy:
Rate (ACC - Accuracy):  84.54267992517475
Corrects:       9018
Substitutions:  1026
Deleted:        113
Inserted:       431
3gram Language Model - Word Accuracy:
Rate (ACC - Accuracy):  90.91267106429063
```

Listado A.20 `05-asr/run.sh` - Script de creación y evaluación de los modelos de un sistema de RAH sobre el subconjunto de validación de VoxForge. Los parámetros de transformación de las distribuciones de probabilidad se han obtenido con el criterio de maximizar valor del índice de precisión léxica (word accuracy) sobre el conjunto de validación.

```
#!/usr/bin/env bash

# Create 3-layer ASR LMMs based on 2-gram & 3-gram Language Models from
# validation sentences and evaluate them.

source scripts/config
# Validation dataset
DUMP="../01-param/valid.dump"
# Acoustic Models
PHONEMES="../02-acoustic/phones.16g.wfsaset"
# Lexical Models
LEXICON="../03-lexicon/lexicon.wfsaset"
# Language Models
LM2G="../04-language/2gram.ktlss"
LM3G="../04-language/3gram.ktlss"
# Initial beam value
BEAM=30
# Desired average trellis size for Automatic Beam Tuning
ATS=200


echo "Building the LMM models"
sautrela LMM -c $LM2G 5 7 $LEXICON 1 -1 $PHONES 0.3 0 > ASR.2-gram.lmm &
sautrela LMM -c $LM3G 7 7 $LEXICON 3 -2 $PHONES 0.2 0 > ASR.3-gram.lmm &
waitAll

echo "Running the decoders"
mkdir rec
sautrela engines/DecodeAndRates.eng -dIn $DUMP -dOut rec/2-gram.dump \
  -beam $BEAM -ats $ATS -i ASR.2-gram.lmm \
  > rec/2-gram.out 2> rec/2-gram.err &
sautrela engines/DecodeAndRates.eng -dIn $DUMP -dOut rec/3-gram.dump \
  -beam $BEAM -ats $ATS -i ASR.3-gram.lmm \
  > rec/3-gram.out 2> rec/3-gram.err &
waitAll

echo "2gram Language Model - Word Accuracy:"
cat rec/2-gram.out
echo "3gram Language Model - Word Accuracy:"
cat rec/3-gram.out
```

```
Corrects:       9271
Substitutions:  624
Deleted:        262
Inserted:       37
```

A.13. Sistema de RAH de gran vocabulario

En las secciones anteriores se han mostrado los pasos a seguir para construir un sistema de RAH de vocabulario reducido (580 palabras). Sin embargo, un sistema de RAH debe ser capaz de afrontar tareas que requieran de un vocabulario significativamente más grande (del orden de decenas de miles de palabras). Como ya se ha explicado previamente, la estructura de la base de datos de VoxForge no ofrece una opción clara para poder evaluar el rendimiento de un sistema de gran vocabulario, ya que el subconjunto de validación consta de 140 frases no pertenecientes a una tarea definida. Sin embargo, el conjunto de entrenamiento sí es parte de lo que puede denominarse una tarea; si revisamos las 43 frases que conforman dicho subconjunto, comprobaremos que todas ellas están extraídas de una novela: *Los argonautas*, de Vicente Blasco Ibáñez. La presente sección muestra los pasos a seguir para construir un sistema de RAH de gran vocabulario (más de 20000 palabras) a partir del texto de la novela *Los argonautas*, y evaluarlo sobre el subconjunto de entrenamiento de VoxForge. Como ha sucedido en el caso anterior, este ejemplo tampoco es del todo real, ya que las señales a evaluar han sido utilizadas para estimar los modelos acústicos. No obstante, este ejercicio servirá para mostrar los pasos a seguir para la creación de un sistema de RAH de gran vocabulario.

El Listado A.21 muestra un script que descarga la novela *Los argonautas* desde `www.gutenberg.org`, extrae[15] el texto y construye un diccionario fonético con el vocabulario resultante. A diferencia del ejemplo anterior, en este caso los silencios son considerados unidades léxicas (una unidad `SILWORD` consta de un único fonema `SIL`). Debido a ello, y para que la probabilidad que el modelo de lenguaje otorgue a la unidad `SILWORD` no sea excesivamente baja, se añade dicha unidad al inicio y final de cada frase. Como resultado de este script, se obtiene un *macrotexto* compuesto por 7547 frases y 193287 palabras, y el tamaño del vocabulario asciende a 20207. El subconjunto de entrenamiento de VoxForge, en el que se validará el sistema, está compuesto a su vez por 43 frases y 681 palabras, y su vocabulario asciende 360 palabras. Todas las palabras del subconjunto de entrenamiento, a excepción de un único término[16], están incluidas en el vocabulario extraído automáticamente de la novela.

A diferencia del caso anterior, es sistema no parte de un conjunto de modelos léxicos, sino que construye un modelo integrado con árbol léxico (`TreeModel`), para lo cual solo se precisa un diccionario fonético y un modelo de lenguaje. El Listado A.22 muestra un script que, partiendo de las frases extraídas para la novela *Los argonautas*, entrena dos modelos de lenguaje de 2-gramas y 3-gramas, respectivamente. Con estos modelos de lenguaje y el diccionario fonético creado anteriormente, genera dos modelos `TreeModel` que integran, en un único WFSA, un modelo de lenguaje de n-gramas y un léxico en árbol.

[15] El filtrado aplicado al texto original trata de extraer conjuntos de frases en minúsculas y sin signos de puntuación. Dicho proceso puede llegar a resultar extremadamente complejo, por lo que el presente código conllevará, muy posiblemente, errores de procesamiento (frases y palabras mal cortadas, borrado de palabras, etc.). No obstante, siempre que el texto resultante dé cobertura suficiente a la tarea, es posible ignorar dichos errores de procesamiento.

[16] El término no visto en entranamiento es `"ventanacon"`. A todas luces, se debe a un error en una de las 43 frases de la propia base de datos de VoxForge, que viene transcrita literalmente como: `"MOVíASE PASANDO DE UNA A OTRA VENTANACON LENTO BALANCEO UNA ESPECIE DE COLUMNA ESBELTA"`.

Listado A.21 `03-lexicon.2/run.sh` - Script de descarga de la novela *Los argonautas* y creación del diccionario fonético. El texto de la novela es descargado desde www.gutenberg.org. Los silencios son considerados una unidad léxica más, y son añadidos al inicio y al final de cada frase para obtener una estimación más robusta del modelo de lenguaje.

```
#!/usr/bin/env bash

# Download the book "Los Argonautas", by Vicente Blasco Ibanez
# from www.gutenberg.org and create a Dictionary
# https://en.wikipedia.org/wiki/Vicente_Blasco_Ib%C3%A1%C3%B1ez
# http://www.gutenberg.org/ebooks/25640

source scripts/config
# download UTF-8 book from gutenberg.org and extract text
function getLosArgonautas {
   wget http://www.gutenberg.org/ebooks/25640.txt.utf-8 -qO /dev/stdout \
     | dos2unix | awk '
      /^Al sentir un roce en el cuello/{w=1}
      /^FIN/{exit}
      w==1 {print}
   ' || true
}
# split text in sencences
function splitSentences { tr "\n" " " | tr '.:?!' "\n" ; }
# clean text
function cleanText {
   sed -e 's/--/ /g' -e 's/[^[:alpha:] ]//g' -e 's/./\L&/g' \
       -e 's/ */ /g' -e 's/^ //' -e 's/ $//'
}
# get phonetic transcription from UTF-8 encoded prompts
function ortho2phon {
   sed -e 's/aequator/aecuator/g' -e 's/vendôme/vendom/g' \
     | iconv -f UTF-8 -t LATIN1 \
     | ../00-database/scripts/ehu_transcribe.LATIN1.pl -w \
     | iconv -f LATIN1 -t UTF-8
}
# split all the character: "inputtext" --> " i n p u t t e x t"
function splitWord { sed -e "s/./ &/g" ; }


echo "Downloadin UTF-8 book from gutenberg.org and extracting the text"
getLosArgonautas > LosArgonautas.utf8.txt

echo "Geting clean sentences (at least 10 words) and adding SILWORD at begin/end"
cat LosArgonautas.utf8.txt | splitSentences | cleanText | awk 'NF>10' \
  | awk '{print "SILWORD",$0,"SILWORD"}' > sentences.txt

echo "Extracting the vocabulary"
cat sentences.txt | tr " " "\12" | grep -v SILWORD | sort | uniq > voc

echo "Transcribing vocabulary"
cat voc | ortho2phon | splitWord | paste -d "" voc - > voc.transc
echo "SILWORD" >> voc
echo "SILWORD SIL" >> voc.transc

echo "Creating the Dictionary"
sautrela Dictionary voc.transc > dictionary
```

Listado A.22 `04-language.2/run.sh` - Script que crea, a partir de las frases extraídas de la novela *Los argonautas*, los modelos `TreeModel` que integran un modelo de lenguaje y un árbol léxico.

```
#!/usr/bin/env bash

# Train 2-gram & 3-gram LM from the book "Los Argonautas" and create TreeModels

source scripts/config
# Training sentences
SENT="../03-lexicon.2/sentences.txt"
# Dictionary
DICT="../03-lexicon.2/dictionary"
# Vocabulary
VOC="../03-lexicon.2/voc"
# Expected vocabulary size (must be higher than the training vocabulary size)
vocSize=$(($(wc -l < $VOC) + 1))


echo "Creating Initial n-gram language models"
sautrela KTLSS -N 2 -v $vocSize > 2gram.ktlss
sautrela KTLSS -N 3 -v $vocSize > 3gram.ktlss

echo "Training the n-gram language models"
sautrela engines/WFSATxtTrainer.eng -txt $SENT -i 2gram.ktlss -o 2gram.ktlss &
sautrela engines/WFSATxtTrainer.eng -txt $SENT -i 3gram.ktlss -o 3gram.ktlss &
waitAll

echo "Building the TreeModels"
sautrela TreeModel 2gram.ktlss $DICT > 2gram.tree &
sautrela TreeModel 3gram.ktlss $DICT > 3gram.tree &
waitAll
```

El Listado A.23 muestra un script que, tras generar dos modelos LMM de dos capas a partir de las dos configuraciones de modelo de lenguaje consideradas, evalúa su rendimiento sobre el subconjunto de entrenamiento de Voxforge. Los valores de las transformaciones de las distribuciones de probabilidad han sido optimizados sobre el subconjunto de 200 mejores envíos de entrenamiento, y los valores del índice de precisión léxico obtenidos son 92,2 % y 95,6 %, para los sistemas basados en 2-gramas y 3-gramas, respectivamente:

```
$ ./run.sh
Building the LMM models
Running the decoders
2gram Language Model - Word Accuracy:
Rate (ACC - Accuracy):  92.15355745682389
Corrects:       29176
Substitutions:  1568
Deleted:        697
Inserted:       202
3gram Language Model - Word Accuracy:
Rate (ACC - Accuracy):  95.55675710060113
```

Listado A.23 `05-asr/run.sh` - Script de creación y evaluación de los modelos de un sistema de RAH de gran vocabulario sobre el subconjunto de entrenamiento de VoxForge. Los parámetros de transformación de las distribuciones de probabilidad han sido obtenidos con el criterio de maximizar el valor del índice de precisión léxica (word accuracy) sobre el conjunto de evaluación.

```
#!/usr/bin/env bash

# Create ASR 2-layer LMMs based on 2-gram & 3-gram TreeModels from the book
# "Los Argonautas" and evaluate them.

source scripts/config
# Train dataset
DUMP="../01-param/train.dump"
# Acoustic Models
PHONEMES="../02-acoustic/phonemes.16g.wfsaset"
# Tree Models
TREE2G="../04-language.2/2gram.tree"
TREE3G="../04-language.2/3gram.tree"
# Initial beam value
BEAM=15
# Desired average trellis size for Automatic Beam Tuning
ATS=200


echo "Building the LMM models"
sautrela LMM -c $TREE2G 2 4 $PHONEMES 0.5 0 > ASR.2-gram.lmm &
sautrela LMM -c $TREE3G 2 5 $PHONEMES 0.5 0 > ASR.3-gram.lmm &
waitAll

echo "Running the decoders"
mkdir rec
sautrela engines/DecodeAndRates.eng -dIn $DUMP -dOut rec/2-gram.dump \
  -beam $BEAM -ats $ATS -i ASR.2-gram.lmm -e SILWORD \
  > rec/2-gram.out 2> rec/2-gram.err &
sautrela engines/DecodeAndRates.eng -dIn $DUMP -dOut rec/3-gram.dump \
  -beam $BEAM -ats $ATS -i ASR.3-gram.lmm -e SILWORD \
  > rec/3-gram.out 2> rec/3-gram.err &
waitAll

echo "2gram Language Model - Word Accuracy:"
cat rec/2-gram.out
echo "3gram Language Model - Word Accuracy:"
cat rec/3-gram.out
```

```
Corrects:       30195
Substitutions:  854
Deleted:        392
Inserted:       151
```

Apéndice B

Sautrela: Comandos

El presente apéndice contiene una descripción detallada de todos los *Comandos* incluidos en Sautrela. Los ejemplos de uso son del tipo:

```
sautrela <nombre_de_comando> <opciones>
```

donde `sautrela`, es un alias de:

```
<path_to_java_command> -jar <path_to_Sautrela.jar>
```

En función del sistema operativo, la instalación de Java™ y la localización del archivo `Sautrela.jar`, la definición de dicho alias variará. El Apéndice A contiene una explicación más detallada sobre la instalación de Sautrela en diferentes sistemas operativos. Nótese que todos los ejemplos del presente apéndice presuponen un entorno de trabajo Unix/Linux con intérprete de comandos Bash.

Como ya se ha indicado en la Sección 3.5, cada comando de Sautrela va asociado a una clase Java, y la línea de comando de Sautrela permite ejecutar un comando a través del nombre completo de clase:

```
$ sautrela path.to.mypackage.MyCommand
```

También es posible acceder al comando a través del nombre sencillo de clase, si ello no da lugar a ambigüedad (es decir, si no existen dos comandos con el mismo nombre sencillo de clase):

```
$ sautrela MyCommand
```

La línea de comando de Sautrela, a través de la opción `-help`, permite acceder también a la página de manual de un comando (una versión reducida de la documentación incluida en el presente apéndice):

```
$ sautrela -help MyCommand

  MyCommand (path.to.mypackage.MyCommand)

    short description of the command

  Syntax:  MyCommand [-options] ARG1 ARG2...

    -option1  description of option1
    -option2  description of option2
    ...
    ARG1  description of ARG1
    ARG2  description of ARG2
    ...
```

La Figura B.1 muestra la estructura general de las páginas de manual de los *Comandos* de Sautrela. En este apéndice, las páginas de manual han sido dispuestas alfabéticamente de acuerdo al nombre sencillo de clase de cada *Comando*.

MyCommand

Nombre

`path.to.mypackage.MyCommand` - Descripción breve del comando.

Sinopsis

```
MyCommand [-options] ARG1 ARG2...
```

Descripción

Descripción pormenorizada del comando.

Opciones y argumentos

Listado y descripción de cada una de las opciones y argumentos del comando.

Ejemplos

Los ejemplos de línea de comando muestran posibles usos prácticos del comando:

```
$ sautrela MyCommand [-options] ARG1 ARG2 ...
```

Vea también

Lista de comandos relacionados.

Figura B.1 Estructura de las páginas de manual de los *Comandos* de Sautrela. Nótese que, a pesar de que todo comando es instanciado a través de la ejecución del comando `sautrela`, por claridad, este último ha sido obviado en la sección *Sinopsis* (no así en la sección *Ejemplos*).

AcousticDataBase

Nombre

`edu.gtts.sautrela.db.AcousticDataBase` - Comprueba y muestra el contenido de una base de datos acústica.

Sinopsis

```
AcousticDataBase [-cnu] [-e enc] [-p list] <-i URL | -r RegExp> URL
```

Descripción

Inspecciona y comprueba el contenido de una base de datos acústica (véase Sección 4.4), enviando a la salida estándar la información relativa al subconjunto de recursos definido mediante un fichero índice (muestra únicamente los recursos cuyo identificador esté contenido en el índice) o mediante una expresión regular (muestra únicamente los recursos cuyo identificador quede representado por la expresión regular). El comando permite seleccionar la información que deseamos extraer (el identificador, el localizador y cada una de las propiedades), así como comprobar que todos los recursos estén accesibles y sean compatibles.

Opciones y argumentos

-c Comprueba el acceso a los recursos de Audio/HTK, generando un mensaje de error si no es posible acceder a alguno de ellos o si se trata de un recurso no compatible. Únicamente se chequea el acceso a los recursos seleccionados, por lo que para poder chequear el acceso a todos los recursos de la base de datos es preciso hacer uso de la opción `-r ".*"` .

-n No muestra el nombre de los recursos seleccionados.

-u No muestra el localizador URL de los recursos seleccionados.

-e enc Codificación utilizada para generar el texto de salida. Si no se especifica, utiliza la codificación por defecto del sistema.

-p list Lista separada por comas de las propiedades a mostrar (véase Sección 4.4). Por defecto, muestra todas. Para no mostrar ninguna propiedad, debe indicarse una lista vacía, `-p ""`.

-i URL Localizador URL de un índice de recursos (un ID de recurso por línea). Se mostrará la información relativa a aquellos recursos cuyo identificador esté contenido en el índice. El orden de salida de los recursos coincidirá con el orden del índice utilizado.

-r RegExp Expresión regular para seleccionar un conjunto de recursos. Se mostrará la información relativa a aquellos recursos cuyo identificador quede representado

por la expresión regular. El orden de salida de los recursos coincidirá con el orden que estos tienen en la base de datos.

URL Localizador URL del índice de la base de datos acústica.

Ejemplos

Para comprobar cada uno de los recursos de la base de datos cuyo fichero índice es `/home/user/MyDatabase/MyAcousticDataBase.xml` y mostrar para cada uno de ellos el valor de la propiedad `Ortho`, pero no el nombre ni el localizador del recurso:

```
$ sautrela AcousticDataBase -cnu -p Ortho -r ".*" \
  /home/user/MyDatabase/MyAcousticDataBase.xml
 Ortho: hasta la vista
 Ortho: mañana toca excursión
 Ortho: pedí otra habitación
 Ortho: hasta la vista
 Ortho: mañana toca excursión
 Ortho: pedí otra habitación
```

Para mostrar toda la información de los recursos cuyos identificadores comiencen por `"wav_ac"` y que se encuentran en la base de datos contenida en el fichero JAR `/home/user/MyDatabase.jar` :

```
$ sautrela AcousticDataBase -r "wav_ac.*" \
  "jar:file://home/user/MyDatabase.jar!/MyAcousticDataBase.xml"
Name: wav_ac_train_001_spk_001
 URL: file:WAV/sentence001.wav
 Phon: a s t a l a b i s t a
 Ortho: hasta la vista
 Speaker: 001
Name: wav_ac_eval_001_spk_013
 URL: file:WAV/sentence002.wav
 Phon: m a h a n a t o k a e s k u r s i o n
 Ortho: mañana toca excursión
 Speaker: 013
```

Para comprobar todos los recursos de una base de datos contenida en un fichero ZIP alojado en un servidor accesible vía HTTP:

```
$ sautrela AcousticDataBase -cnup "" -r ".*" \
  "jar:http://my.server.com/pub/MyDatabase.zip!/MyAcousticDataBase.xml"
```

Vea también

AudioResource, HTKResource

AudioResource

Nombre

`edu.gtts.sautrela.audio.AudioResource` - Muestra la información relativa al formato de un conjunto de recursos de audio.

Sinopsis

```
AudioResource URL ...
```

Descripción

Analiza un conjunto de recursos de audio a través de sus localizadores y muestra la información relativa a su formato (número de canales, codificación, frecuencia de muestreo, tamaño de muestra, etc.). El comando genera un mensaje error en caso de no poder acceder al recurso bien porque no esté accesible, bien porque el esquema del localizador URL no esté soportado (véase Sección 3.7). También se genera un mensaje de error si el formato del recurso no es compatible (véase Sección 4.2). Sautrela soporta ficheros en formato AIFF, AU y WAV, audio de 8 y 16 bits mono y estéreo, frecuencias de muestreo entre 8 kHz y 48 kHz, y codificaciones lineal, a-law y mu-law.

Opciones y argumentos

URL Localizador URL del recurso de audio a analizar.

Ejemplos

Para analizar todos los ficheros de audio del directorio actual[1]:

```
$ shopt -s nullglob
$ sautrela AudioResource *.{aiff,wav,au}
Resource:  file:file1.wav
  Channels 2
  Encoding PCM_SIGNED
  SampleRate 44100.0
  SampleSize 16
  Samples 11414016
Resource:  file:file2.wav
  Channels 1
  Encoding PCM_SIGNED
  SampleRate 16000.0
  SampleSize 16
```

[1] El comando `shopt -s nullglob` configura la expansión *glob* del shell para que en caso de que el patrón no corresponda con ningún fichero, este no sea expandido a su valor literal. Por defecto, un patrón del tipo `*.ext` que no coincida con ninguno de los ficheros es expandido a su literal `"*.ext"` (lo que seguramente generará un error en la ejecución del comando), mientras que con la configuración `nullglob` no se genera ningún resultado (evitando posibles errores de ejecución del comando).

```
  Samples 4141140
Resource:  file:file3.wav
  Channels 1
  Encoding PCM_SIGNED
  SampleRate 8000.0
  SampleSize 16
  Samples 2070570
Resource:  file:file4.au
  Channels 1
  Encoding ULAW
  SampleRate 16000.0
  SampleSize 8
  Samples 4141140
```

Vea también

AcousticDataBase, HTKResource

CHMM

Nombre

`edu.gtts.sautrela.wfsa.models.CHMM` - Crea un HMM continuo.

Sinopsis

```
CHMM [-E] [-n name] [ -d gDim | -g URL ] size
```

Descripción

Vuelca a la salida estándar un modelo oculto de Markov (HMM, por sus siglas en inglés) continuo (con emisiones en un espacio continuo) con el nombre y número de estados indicado, usando bien una topología lineal con bucles, bien una topología ergódica. Cada estado lleva asociado un modelo de mezcla de Gaussianas (GMM, por sus siglas en inglés), el cual puede ser creado a partir de un GMM existente (a cada estado se le asigna una copia del GMM cargado) o inicializado con una única componente de medias cero y varianzas unidad (la implementación `GMM` de Sautrela hace uso de matrices de covarianza diagonal, véase el comando GMM). Cada Gaussiana establece además un valor mínimo a las varianzas (*varFloor*), que es fijado independientemente para cada una de las dimensiones a partir de su primera estimación ($varFloor = var_{ini}/100$). Las probabilidades correspondientes a la topología (probabilidades iniciales, finales y de transición) son inicializadas de acuerdo a una distribución uniforme. Todos los parámetros de entrenamiento del modelo resultante, tanto los relativos a su topología (entrenamiento de las probabilidades de transición) como los relativos a las componentes Gaussianas (entrenamiento de los pesos, las medias y las varianzas) quedan activados.

Opciones y argumentos

-E Hace uso de una topología ergódica (todos los estados son iniciales y finales, pudiendo transitar de manera equiprobable entre cualesquiera dos estados). La topología por defecto es lineal con bucles (todo estado solo puede transitar sobre sí mismo o al siguiente estado, siendo el primer estado el único inicial, y el último estado el único final). A fin de romper la simetría de una topología ergódica, se añade una leve perturbación[2] a las probabilidades uniformes de transición, permitiendo así que las re-estimaciones de los parámetros no terminen siendo iguales para todos los estados.

-n name Nombre del HMM resultante [`noName` por defecto].

-d gDim Dimensión de las Gaussianas [`1` por defecto]. Cada estado contará con una única componente inicial de dimensión `gDim`. Todos los parámetros de entrenamiento de los GMMs creados quedarán activados.

[2] Los valores de la distribución de probabilidad son inicializados a `100`, se les suma un valor aleatorio en el rango $[0, 1)$ y son finalmente re-normalizados.

-g URL Localizador del GMM a utilizar como semilla (inicialización) de cada estado. Todos los parámetros de entrenamiento de los GMMs quedarán activados (los parámetros de entrenamiento del GMM cargado son descartados).

size Número de estados.

Ejemplos

Para generar un HMM de tres estados y topología lineal con GMMs iniciales de dimensión 10:

```
$ sautrela CHMM -n example -d 10 3
<?xml version="1.0" encoding="UTF-8" ?>
<WFSA name="example" trainTrans="true" trainWeight="true" trainMean="true"
     trainVar="true" splitIfOrphan="true" className="edu.gtts.sautrela.
    wfsa.models.CHMM">
  <IniState name="0" linProb="1.0"/>
  <FinState name="2" linProb="0.5"/>
  <Transition from="0" to="0" linProb="0.5"/>
  <Transition from="0" to="1" linProb="0.5"/>
  <Transition from="1" to="1" linProb="0.5"/>
  <Transition from="1" to="2" linProb="0.5"/>
  <Transition from="2" to="2" linProb="0.5"/>
  <State name="0">
    <Gaussian name="example-s0-g0" weight="1.0">
      <Mean> 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 </Mean>
      <Var> 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 </Var>
      <VarFloor> 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 </VarFloor>
    </Gaussian>
  </State>
  <State name="1">
    <Gaussian name="example-s1-g0" weight="1.0">
      <Mean> 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 </Mean>
      <Var> 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 </Var>
      <VarFloor> 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 </VarFloor>
    </Gaussian>
  </State>
  <State name="2">
    <Gaussian name="example-s2-g0" weight="1.0">
      <Mean> 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 </Mean>
      <Var> 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 </Var>
      <VarFloor> 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 </VarFloor>
    </Gaussian>
  </State>
</WFSA>
```

Para generar un HMM de cuatro estados y topología lineal a partir de un GMM contenido en el fichero `mixture.gmm`:

```
$ sautrela CHMM -n myCHMM -g mixture.gmm 4 > myErgodic.chmm
```

Para generar un HMM ergódico de 2 estados con GMMs iniciales de dimensión 6. La perturbación aplicada a las probabilidades uniformes de transición evita la simetría propia de una topología ergódica:

```
$ sautrela CHMM -En myErgodicCHMM -d 6 2
<?xml version="1.0" encoding="UTF-8" ?>
<WFSA name="myErgodicCHMM" trainTrans="true" trainWeight="true" trainMean
    ="true" trainVar="true" splitIfOrphan="true" className="edu.gtts.
    sautrela.wfsa.models.CHMM">
  <IniState name="0" linProb="0.500228198393659"/>
  <IniState name="1" linProb="0.4997718016063406"/>
  <FinState name="0" linProb="0.3343003823279776"/>
  <FinState name="1" linProb="0.3333100975199566"/>
  <Transition from="0" to="0" linProb="0.332673870382267"/>
  <Transition from="0" to="1" linProb="0.3330257472897554"/>
  <Transition from="1" to="0" linProb="0.3345905163079629"/>
  <Transition from="1" to="1" linProb="0.3320993861720802"/>
  <State name="0">
    <Gaussian name="myErgodicCHMM-s0-g0" weight="1.0">
      <Mean> 0.0 0.0 0.0 0.0 0.0 0.0 </Mean>
      <Var> 1.0 1.0 1.0 1.0 1.0 1.0 </Var>
      <VarFloor> 0.0 0.0 0.0 0.0 0.0 0.0 </VarFloor>
    </Gaussian>
  </State>
  <State name="1">
    <Gaussian name="myErgodicCHMM-s1-g0" weight="1.0">
      <Mean> 0.0 0.0 0.0 0.0 0.0 0.0 </Mean>
      <Var> 1.0 1.0 1.0 1.0 1.0 1.0 </Var>
      <VarFloor> 0.0 0.0 0.0 0.0 0.0 0.0 </VarFloor>
    </Gaussian>
  </State>
</WFSA>
```

Vea también

CHMMEdit, CHMMSetEdit, DHMM, GMM

CHMMEdit

Nombre

`edu.gtts.sautrela.wfsa.models.CHMMEdit` - Modifica un HMM continuo.

Sinopsis

```
CHMMEdit [-mMsStTvVwW] [-n name] [-g gNum] URL
```

Descripción

Toma un HMM continuo y vuelca a la salida estándar una versión modificada del mismo. Permite modificar el nombre, el número de componentes por estado y los parámetros de entrenamiento, tanto los relativos a su topología (activación/desactivación del entrenamiento de las probabilidades de transición) como los relativos a las componentes Gaussianas (activación/desactivación del entrenamiento de los pesos, las medias y las varianzas).

Opciones y argumentos

-m Se desactiva el entrenamiento de medias.

-M Se activa el entrenamiento de medias [activado por defecto].

-s Se desactiva la eliminación y desdoblamiento ante Gaussianas huérfanas.

-S Se activa la eliminación y desdoblamiento ante Gaussianas huérfanas [activado por defecto]. Aquellas Gaussianas cuyo peso resulte inferior a $1/(100 \cdot g)$ (siendo g el número de componentes del GMM) son eliminadas, realizándose un desdoblamiento (*splitting*) de la Gaussiana más probable.

-t Se desactiva el entrenamiento de transiciones.

-T Se activa el entrenamiento de transiciones [activado por defecto].

-v Se desactiva el entrenamiento de varianzas.

-V Se activa el entrenamiento de varianzas [activado por defecto].

-w Se desactiva el entrenamiento de pesos.

-W Se activa el entrenamiento de pesos [activado por defecto].

-n name Modifica el nombre del modelo resultante.

-g gNum Número de Gaussianas. La reducción de Gaussianas se realiza mediante la eliminación de las Gaussianas menos probables, repartiendo su peso a partes iguales entre el resto de Gaussianas. Por el contrario, el incremento de Gaussianas se realiza mediante el desdoblamiento aleatorio (*splitting*) de la Gaussiana de

mayor peso. Dicho desdoblamiento se obtiene generando un vector aleatorio de medias en torno al vector de medias original, haciendo uso de la varianza mínima. La varianza y la varianza mínima de la nueva Gaussiana son idénticas a las originales y el peso es compartido a partes iguales por ambas Gaussianas (la Gaussiana original ve reducido su peso a la mitad).

URL Localizador del HMM a utilizar como inicialización o "semilla". El modelo resultante es redireccionado a la salida estándar.

Ejemplos

Para Tomar un HMM continuo contenido en el fichero `phone.32g.chmm`, aumentar a 64 el número de componentes por estado y desactivar el entrenamiento de las transiciones:

```
$ sautrela CHMMEdit -tg 64 phone.32g.chmm > phone.64g.chmm
```

Para crear, a partir de una mezcla de Gaussianas contenidas en el fichero `UBM.gmm`, un HMM ergódico de 5 estados que solo tenga activado el entrenamiento de transiciones y pesos. Dado que al crear un HMM todos los parámetros de entrenamiento quedan activados por defecto, resulta necesario crear un modelo inicial y modificar sus parámetros de entrenamiento posteriormente. El siguiente ejemplo hace uso de una tubería (*pipe*) para no tener que utilizar un almacenamiento intermedio temporal para el modelo inicial:

Dado que al crear un HMM continuo todos los parámetros de re-estimación quedan activados por defecto, para poder obtener un HMM que tenga desactivado alguno de ellos, resulta necesario crear un modelo inicial y modificar sus parámetros posteriormente. El siguiente ejemplo crea, a partir de una mezcla de Gaussianas contenidas en el fichero `UBM.gmm`, un HMM ergódico de 5 estados que solo tenga activado la reestimación de transiciones y pesos. Además, hace uso del mecanismo de substitución de procesos (véase Sección 4.1.2) para no tener que almacenar temporalmente el modelo inicial:

```
$ sautrela CHMMEdit -msTvW <(sautrela CHMM -n ErgodicCHMM -Eg UBM.gmm 5) \
    > ergodyc.chmm
```

Vea también

CHMM, CHMMSetEdit, GMM

CHMMSetEdit

Nombre

`edu.gtts.sautrela.wfsa.models.CHMMSetEdit` - Modifica un conjunto de HMMs continuos.

Sinopsis

```
CHMMSetEdit [-mMtsSTvVwW] [-n names] [-g gNum] URL
```

Descripción

Toma un conjunto de `CHMM`s y vuelca a la salida estándar una versión modificada del mismo. Permite, de forma análoga al comando CHMMEdit, modificar el número de componentes y los parámetros de entrenamiento de un conjunto de HMMs continuos, así como replicar y renombrar los modelos. Este comando permite realizar cuatro tareas principales:

- **Modificar la configuración de entrenamiento de un conjunto de `CHMM`s.** Es posible habilitar o deshabilitar el entrenamiento de las transiciones, los pesos, las medias y las varianzas de las Gaussianas de los modelos. Por defecto, todos los parámetros de entrenamiento de un `CHMM` están activados.

- **Incrementar el número de Gaussianas de un conjunto de `CHMM`s.** En el proceso de entrenamiento de los HMMs continuos puede resultar adecuado partir de un número inicial de componentes Gaussianas e ir aumentándolo de forma controlada. Una práctica común es la de inicializar los modelos con una sola Gaussiana y re-entrenarlos hasta cumplir algún criterio de convergencia, para después duplicar su número de componentes y volver a repetir el proceso tantas veces como se desee.

- **Crear un conjunto de `CHMM`s a partir de un `CHMM` prototipo.** El conjunto inicial de HMMs continuos puede ser inicializado a partir de un conjunto que contenga un único modelo prototipo que es replicado.

- **Crear un conjunto de `CHMM`s especializado (mayor número de modelos) a partir de otro más general (menor número de modelos).** Se trata de una generalización del caso anterior. Se parte de un conjunto de modelos que representa un conjunto de categorías, para obtener un conjunto de modelos mayor (que representa un conjunto más complejo de categorías). Cada modelo original es replicado tantas veces como nuevas categorías correspondan a su categoría original. De manera análoga al incremento gradual de las componentes Gaussianas, la especialización progresiva de los modelos permite partir de estimaciones más robustas, mejorando la convergencia de los modelos estimados.

Opciones y argumentos

-m Se desactiva el entrenamiento de medias.

-M Se activa el entrenamiento de medias [activado por defecto].

-s Se desactiva la eliminación y desdoblamiento ante Gaussianas huérfanas.

-S Se activa la eliminación y desdoblamiento ante Gaussianas huérfanas [activado por defecto]. Aquellas Gaussianas cuyo peso resulte demasiado pequeño son eliminadas, realizándose un desdoblamiento de la Gaussiana más probable.

-t Se desactiva el entrenamiento de transiciones.

-T Se activa el entrenamiento de transiciones [activado por defecto].

-v Se desactiva el entrenamiento de varianzas.

-V Se activa el entrenamiento de varianzas [activado por defecto].

-w Se desactiva el entrenamiento de pesos.

-W Se activa el entrenamiento de pesos [activado por defecto].

-n names Lista de nombres (separados por ",") de los modelos resultantes. Por defecto, el conjunto resultante conserva el conjunto de HMMs originales, pero en caso de aportar una lista de nombres, el tamaño del conjunto resultante podrá resultar modificado. La secuencia de HMMs del conjunto de referencia (en el orden indicado en el descriptor XML, véase el comando WFSASet) es utilizada para generar modelos equivalentes a los que se les asignará el nombre correspondiente de la lista. Si el tamaño de la lista de nombres es inferior al tamaño del conjunto de referencia, se descartan modelos del conjunto de referencia. Por el contrario, si el tamaño de la lista es mayor, se volverán a reutilizar cíclicamente los modelos del conjunto de referencia, tantas veces como sea preciso. Si la lista contiene nombres repetidos (todos los modelos de un `WFSASet` deben tener nombres distintos), se descarta el HMM de referencia correspondiente y se pasa al siguiente elemento de la lista.

-g gNum Número de Gaussianas. La reducción de Gaussianas se realiza mediante la eliminación de las Gaussianas menos probables, repartiendo su peso a partes iguales entre el resto de Gaussianas. Por el contrario, el incremento de Gaussianas se realiza mediante el desdoblamiento aleatorio (*splitting*) de la Gaussiana de mayor peso. Dicho desdoblamiento se obtiene generando un punto aleatorio en torno a la media original, haciendo uso de la varianza mínima (en caso de no haberse estimado aún, se hace uso de la varianza). La varianza y la varianza mínima de la nueva Gaussiana son idénticas a las originales y el peso es compartido a partes iguales por ambas Gaussianas (la Gaussiana original ve reducido su peso a la mitad).

URL Localizador del conjunto a modificar. El conjunto resultante es redireccionado a la salida estándar

Ejemplos

Un GMM (incluso de una única componente) estimado a partir del conjunto de muestras de entrenamiento (fichero `ubm.1.gmm`) es un buen punto de partida para generar un CHMM inicial o prototipo (fichero `proto.1g.chmm`) con el que componer un conjunto que contenga únicamente dicho modelo prototipo (fichero `proto.1g.chmm`set). Partiendo de este conjunto, podemos crear un conjunto inicial de modelos para las cinco vocales y el silencio:

```
$ sautrela CHMM -n proto -g ubm.1.gmm 3 > proto.1g.chmm
$ sautrela WFSASet -n protoSet proto.1g.chmm > proto.1g.chmmset
$ sautrela CHMMSetEdit -n a,e,i,o,u,sil proto.1g.chmmset \
  > vowels.1g.chmmset
```

Tras sucesivas fases de re-estimación, es momento de aumentar el número de componentes Gaussianas de los modelos y volver a re-estimarlos sucesivamente, hasta obtener el número de componentes deseado:

```
$ sautrela CHMMSetEdit -g 2 vowels.1g.chmmset > vowels.2g.chmmset
$ ...
$ sautrela CHMMSetEdit -g 4 vowels.2g.chmmset > vowels.4g.chmmset
$ ...
$ sautrela CHMMSetEdit -g 8 vowels.4g.chmmset > vowels.8g.chmmset
$ ...
$ sautrela CHMMSetEdit -g 16 vowels.8g.chmmset > vowels.16g.chmmset
```

Una vez conseguido el número de componentes deseado, es posible generar modelos más especializados considerando, por ejemplo, diferentes contextos de aparición de las vocales. Supongamos que deseamos modelar tres posibles contextos para cada una de las vocales, pero no así para el silencio, que mantendría un único modelo. Podríamos además desactivar el re-entrenamiento de las varianzas:

```
$ sautrela CHMMSetEdit \
  -vn a1,e1,i1,o1,u1,sil,a2,e2,i2,o2,u2,sil,a3,e3,i3,o3,u3 \
  vowels.16g.chmmset > vowels.16g_3ctxt.chmmset
```

Vea también

CHMM, CHMMEdit, WFSASet

CommandNavigator

Nombre

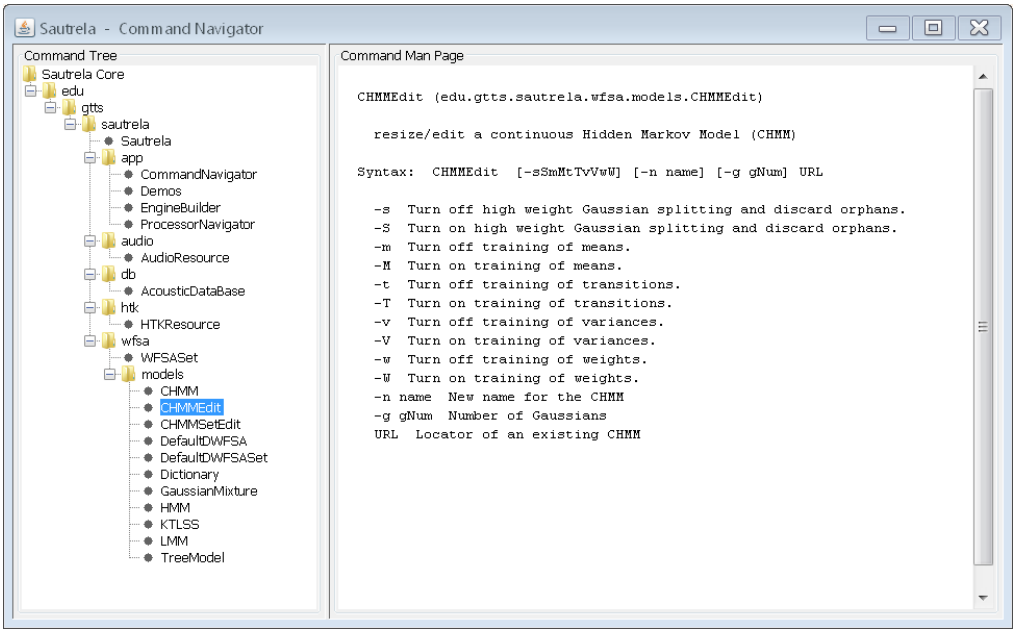
`edu.gtts.sautrela.app.CommandNavigator` - Aplicación gráfica que muestra el conjunto de *Comandos* contenidos en plugins propios y de terceros.

Sinopsis

```
CommandNavigator
```

Descripción

Ejecuta la interfaz gráfica que permite consultar el conjunto de *Comandos* contenidos en plugins propios y de terceros, así como su correspondiente documentación:



Vea también

Demos, EngineBuilder, ProcessorNavigator, Sutrela

DefaultDWFSA

Nombre

`edu.gtts.sautrela.wfsa.models.DefaultDWFSA` - Crea un autómata de estados finitos ponderado (Weigthed Finite State Automaton, WFSA) determinista a partir de una transcripción.

Sinopsis

```
DefaultDWFSA [-n name] [-lri -s symbol] symbol ...
```

Descripción

Vuelca a la salida estándar un autómata de estados finitos ponderado (Weigthed Finite State Automaton, WFSA) determinista a partir de una transcripción. El autómata generado emitirá la secuencia de símbolos dada en la línea de comandos, pudiendo emitir de forma opcional símbolos extras (típicamente símbolos de relleno o silencio) al inicio y final de la transcripción, así como entre cada uno de los símbolos que forman la transcripción.

Opciones y argumentos

-n name Nombre del autómata resultante [`noName` por defecto].

-lri Permite la emisión opcional de un símbolo especial (opción `-s symbol`) al inicio de la transcripción (`-l`, *left*), al final de la transcripción (`-r`, *right*) o entre los símbolos de la transcripción (`-i`, *inner*).

-s symbol El símbolo especial que podrá ser emitido en función de la máscara determinada por las opciones `-lri`.

symbol Símbolos de la transcripción.

Ejemplos

Para generar un autómata determinista que sirva de modelo léxico para la palabra *cacharro*:

```
$ sautrela DefaultDWFSA -n cacharro k a X a R o
<?xml version="1.0" encoding="UTF-8" ?>
<WFSA name="cacharro" className="edu.gtts.sautrela.wfsa.models.
    DefaultDWFSA">
  <IniState name="0"/>
  <FinState name="6" linProb="1.0"/>
  <Transition from="0" to="1" symbol="k" linProb="1.0"/>
  <Transition from="1" to="2" symbol="a" linProb="1.0"/>
  <Transition from="2" to="3" symbol="X" linProb="1.0"/>
  <Transition from="3" to="4" symbol="a" linProb="1.0"/>
```

```
  <Transition from="4" to="5" symbol="R" linProb="1.0"/>
  <Transition from="5" to="6" symbol="o" linProb="1.0"/>
</WFSA>
```

Podemos generar un autómata determinista que sirva de modelo léxico para la palabra *cacharro*, pero con posibilidad de emitir un silencio (símbolo `SIL`) al final. Una vez emitido el simbolo `"o"` del final de la transcripción `"k a X a R o"`, el autómata se encuentra en el estado final denominado `"6"`, el cual tiene una probabilidad `0.5` tanto de finalizar, como de transitar al estado final `"i6"` emitiendo el símbolo especial `SIL`:

```
$ sautrela DefaultDWFSA -n cacharro -rs SIL k a X a R o
<?xml version="1.0" encoding="UTF-8" ?>
<WFSA name="cacharro" className="edu.gtts.sautrela.wfsa.models.
    DefaultDWFSA">
  <IniState name="0"/>
  <FinState name="6" linProb="0.5"/>
  <FinState name="i6" linProb="1.0"/>
  <Transition from="0" to="1" symbol="k" linProb="1.0"/>
  <Transition from="1" to="2" symbol="a" linProb="1.0"/>
  <Transition from="2" to="3" symbol="X" linProb="1.0"/>
  <Transition from="3" to="4" symbol="a" linProb="1.0"/>
  <Transition from="4" to="5" symbol="R" linProb="1.0"/>
  <Transition from="5" to="6" symbol="o" linProb="1.0"/>
  <Transition from="6" to="i6" symbol="SIL" linProb="0.5"/>
</WFSA>
```

Para generar un autómata determinista que sirva de modelo de transcripción fonética para la locución *hace buen tiempo*, con posibilidad de emisión inicial y final de un silencio:

```
$ sautrela DefaultDWFSA -n "hace buen tiempo" \
  -lrs SIL a z e b u e n t i e m p o > sentence.wfsa
```

Para generar un autómata determinista que represente la secuencia de palabras que da lugar a un comando de voz, con posibilidad de contener silencios al inicio, al final y entre las palabras (en este caso, el silencio es considerado una unidad léxica):

```
$ sautrela DefaultDWFSA -n comando1 -lris SIL  iniciar navegador web \
  > comando1.wfsa
```

Vea También

DefaultDWFSASet, WFSASet

DefaultDWFSASet

Nombre

`edu.gtts.sautrela.wfsa.models.DefaultDWFSASet` - Crea un conjunto de autómatas de estados finitos ponderados (Weigthed Finite State Automata, WFSA) deterministas a partir de un diccionario.

Sinopsis

```
DefaultDWFSASet [-n name] [-ilr -s symbol] URL
```

Descripción

Vuelca a la salida estándar un conjunto de autómatas de estados finitos ponderados deterministas a partir de un diccionario (véase el comando Dictionary). Genera un autómata determinista (un modelo `DefaultDWFSA`) para cada una de las entradas del diccionario, de manera que cada autómata emitirá la secuencia correspondiente a su transcripción, pudiendo opcionalmente emitir símbolos extras (típicamente símbolos de relleno o silencio) al inicio y final de la transcripción, así como entre cada uno de los símbolos de la misma.

Opciones y argumentos

-n name Nombre del conjunto de autómatas resultante [`noName` por defecto].

-lri Permite la emisión opcional de un símbolo especial (opción `-s symbol`) al inicio de la transcripción (`-l`), al final de la transcripción (`-r`) o entre los símbolos de la transcripción (`-i`).

-s symbol El símbolo especial que podrá ser emitido en función de la máscara determinada por las opciones `-lri`.

URL Localizador del diccionario. El diccionario no puede contener transcripciones múltiples, por lo que, de contenerlas, se genera un mensaje de error.

Ejemplos

Partiendo del fichero de texto `trans.txt` que contiene el conjunto de transcripciones correspondientes a un léxico (una transcripción por línea, véase el comando Dictionary), el comando que sigue genera su correspondiente diccionario, y a partir de este, genera un conjunto de autómatas deterministas que permiten inserciones del símbolo `SIL` al final de cada palabra:

```
$ sautrela Dictionary trans.txt > trans.dict
$ sautrela DefaultDWFSASet -n myLexicon -rs SIL trans.dict \
  > lexicon.wfsaset
```

Vea también

DefaultDWFSA, Dictionary, WFSASet

Demos

Nombre

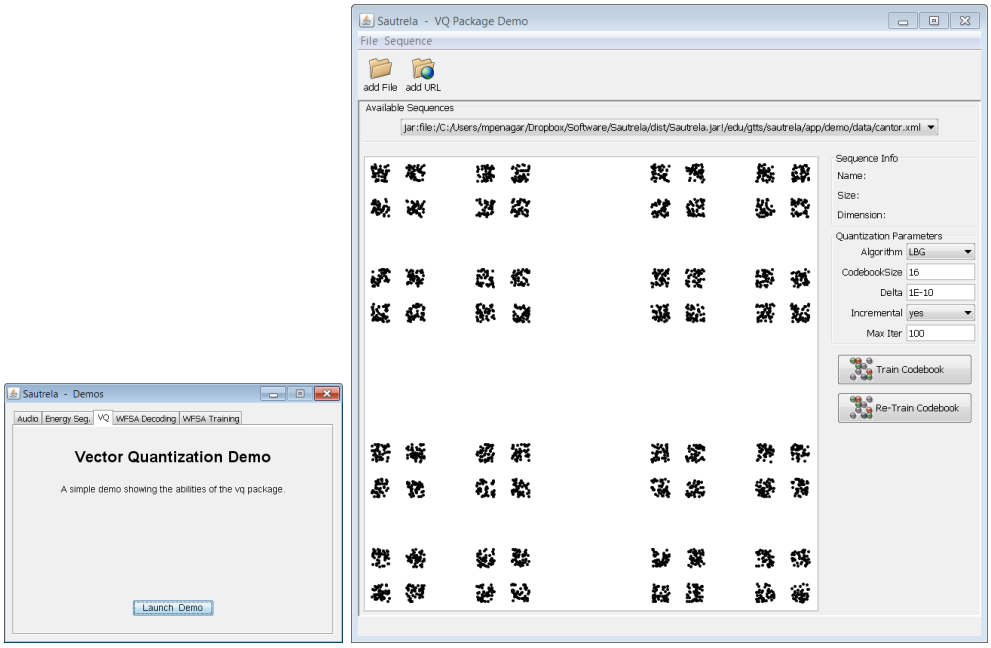
`edu.gtts.sautrela.app.Demos` - Aplicación gráfica que muestra el conjunto de demos contenido en Sautrela.

Sinopsis

```
Demos
```

Descripción

Ejecuta la interfaz gráfica que da acceso al conjunto de demos que muestran diversas funcionalidades de Sautrela:



Vea también

CommandNavigator, EngineBuilder, ProcessorNavigator, Sutrela

DHMM

Nombre

`edu.gtts.sautrela.wfsa.models.DHMM` - Crea un HMM discreto.

Sinopsis

```
DHMM [-E] [-n name] [-r from,to] [-s symbol[,symbol ...]] size
```

Descripción

Vuelca a la salida estándar un modelo oculto de Markov (HMM, por sus siglas en inglés) discreto (emisiones discretas) con el nombre y número de estados indicado, usando bien una topología lineal con bucles, bien una topología ergódica. El alfabeto de emisión puede ser suministrado bien mediante un rango de números enteros, bien explícitamente (o de ambas maneras). Todas las probabilidades de emisión y las correspondientes a la topología (probabilidades iniciales, finales y de transición) son inicializadas de acuerdo a una distribución uniforme.

Opciones y argumentos

-E Hace uso de una topología ergódica (todos los estados son iniciales y finales, pudiendo transitar de manera equiprobable entre cualesquiera dos estados). La topología por defecto es lineal con bucles (todo estado solo puede transitar sobre sí mismo o al siguiente estado, siendo el primer estado el único inicial, y el último estado el único final). A fin de romper la simetría de una topología ergódica, se añade una leve perturbación[3] a las probabilidades uniformes de transición, permitiendo así que las re-estimaciones de los parámetros no terminen siendo iguales para todos los estados.

-n name Nombre del HMM resultante.

-r from,to Rango entero de símbolos de emisión. Cada estado tendrá una distribución uniforme asociada al conjunto completo de símbolos de emisión. Puede ser usada junto a la opción `-s`.

-s symbol[,symbol ...] Lista separada por comas del conjunto de símbolos a emitir. Cada estado tendrá una distribución uniforme asociada al conjunto completo de símbolos de emisión. Puede ser usada junto a la opción `-r`.

size Número de estados.

[3] Los valores de la distribución de probabilidad son inicializados a `100`, se les suma un valor aleatorio en el rango $[0, 1)$ y son finalmente re-normalizados.

Ejemplos

Para generar un HMM discreto, con topología lineal de tres estados y con un alfabeto de emisión $A = \{1, 2, 3, 4\}$:

```
$ sautrela DHMM -n myDHMM -r 1,4 3
<?xml version="1.0" encoding="UTF-8" ?>
<WFSA name="myDHMM" size="3" cdbkSize="4" className="edu.gtts.sautrela.
    wfsa.models.DHMM">
  <IniState name="0" linProb="1.0"/>
  <FinState name="2" linProb="1.0"/>
  <Transition from="0" to="0" linProb="0.5"/>
  <Transition from="0" to="1" linProb="0.5"/>
  <Transition from="1" to="1" linProb="0.5"/>
  <Transition from="1" to="2" linProb="0.5"/>
  <Emision state="0" symbol="1" linProb="0.25"/>
  <Emision state="0" symbol="2" linProb="0.25"/>
  <Emision state="0" symbol="3" linProb="0.25"/>
  <Emision state="0" symbol="4" linProb="0.25"/>
  <Emision state="1" symbol="1" linProb="0.25"/>
  <Emision state="1" symbol="2" linProb="0.25"/>
  <Emision state="1" symbol="3" linProb="0.25"/>
  <Emision state="1" symbol="4" linProb="0.25"/>
  <Emision state="2" symbol="1" linProb="0.25"/>
  <Emision state="2" symbol="2" linProb="0.25"/>
  <Emision state="2" symbol="3" linProb="0.25"/>
  <Emision state="2" symbol="4" linProb="0.25"/>
</WFSA>
```

Para generar un modelo ergódico de dos estados y alfabeto $A = \{X, Y, Z\}$. Las probabilidades de transición sufren una leve perturbación que rompe la simetría:

```
$ sautrela DHMM -En myErgodicDHMM -s X,Y,Z 2
<?xml version="1.0" encoding="UTF-8" ?>
<WFSA name="myErgodicDHMM" size="2" cdbkSize="3" className="edu.gtts.
    sautrela.wfsa.models.DHMM">
  <IniState name="0" linProb="0.49978859441188966"/>
  <IniState name="1" linProb="0.5002114055881106"/>
  <FinState name="0" linProb="0.3342825837342718"/>
  <FinState name="1" linProb="0.33309712335891806"/>
  <Transition from="0" to="0" linProb="0.3332680405349328"/>
  <Transition from="0" to="1" linProb="0.33244937573079525"/>
  <Transition from="1" to="0" linProb="0.3341014106803478"/>
  <Transition from="1" to="1" linProb="0.33280146596073407"/>
  <Emission state="0" symbol="X" linProb="0.3333333333333333"/>
  <Emission state="0" symbol="Y" linProb="0.3333333333333333"/>
  <Emission state="0" symbol="Z" linProb="0.3333333333333333"/>
  <Emission state="1" symbol="X" linProb="0.3333333333333333"/>
  <Emission state="1" symbol="Y" linProb="0.3333333333333333"/>
  <Emission state="1" symbol="Z" linProb="0.3333333333333333"/>
```

```
</WFSA>
```

Vea también

DefaultDWFSA, CHMM, WFSASet

Dictionary

Nombre

`edu.gtts.sautrela.wfsa.models.Dictionary` - Crea un un diccionario de transcripciones.

Sinopsis

```
Dictionary [-e encoding] URL
```

Descripción

Vuelca a la salida estándar un diccionario de transcripciones a partir de un recurso de texto que contenga dichas transcripciones. El diccionario resultante es un sencillo documento XML que elimina los problemas de codificación asociados al recurso de texto original. Las entradas del diccionario pueden contener transcripciones múltiples.

Opciones y argumentos

-e encoding Codificación del recurso de texto original. Si no se especifica, utiliza la codificación por defecto del sistema.

URL Localizador del recurso de texto que contiene las transcripciones. Cada línea de texto debe contener una palabra y su correspondiente transcripción delimitada por espacios. Una palabra puede contener múltiples transcripciones (una por línea).

Ejemplos

Partiendo del fichero de transcripciones `trans.txt` que contiene las siguientes transcripciones:

```
CASA k a s a
SALUD s a l u d
CAZA k a z a
SALUD s a l u z
```

el siguiente comando genera un diccionario de transcripciones :

```
$ sautrela Dictionary trans.txt
<?xml version="1.0" encoding="UTF-8" ?>
<Dictionary>
  <Key name="CASA">
    <Value value="k a s a"/>
  </Key>
  <Key name="CAZA">
    <Value value="k a z a"/>
```

```
  </Key>
  <Key name="SALUD">
    <Value value="s a l u d"/>
    <Value value="s a l u z"/>
  </Key>
</Dictionary>
```

Vea también

DefaultDWFSASet, TreeModel

EngineBuilder

Nombre

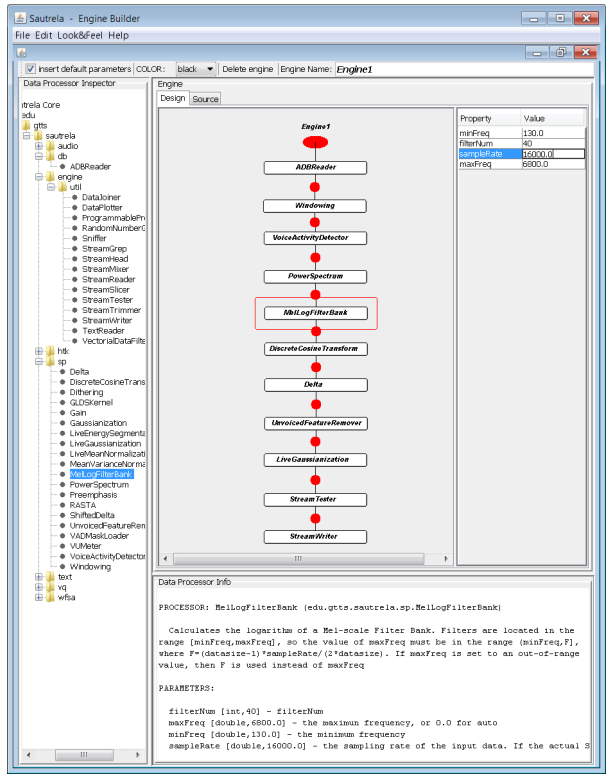
`edu.gtts.sautrela.app.EngineBuilder` - Aplicación gráfica para crear y modificar *Engines*

Sinopsis

```
EngineBuilder
```

Descripción

Ejecuta la interfaz gráfica que permite la creación y modificación de engines. Cualquier *Procesador* puede ser arrastrado y conectado a la línea de procesamiento de la *Engine*, siendo posible modificar cada uno de sus parámetros así como los Buffers que los interconectan. La aplicación importa todos los plugins, tanto propios como de terceros, y muestra en todo momento la documentación del procesador seleccionado:



Vea también

CommandNavigator, Demos, ProcessorNavigator, Sutrela

GMM

Nombre

`edu.gtts.sautrela.wfsa.models.GMM` - Crea o modifica un GMM.

Sinopsis

```
GMM [-mMsSvVwW] [-n name] [-g gNum] [-d gDim] [URL]
```

Descripción

Crea o carga un modelo de mezcla de Gaussianas (GMM, por sus siglas en inglés) y vuelca a la salida estándar una versión modificada del mismo. Un GMM se compone de una mezcla ponderada de Gaussianas multivariantes, cada una de ellas con sus correspondientes vectores de media y varianza (la implementación GMM de Sautrela hace uso de matrices de covarianza diagonal). Cada Gaussiana establece además un valor mínimo a las varianzas (*varFloor*), que es fijado independientemente para cada una de las dimensiones a partir de su primera re-estimación ($varFloor = var_{ini}/100$). De este modo se evita que las varianzas de las Gaussianas poco pobladas tiendan a cero, lo que provocaría problemas numéricos. Existe además un mecanismo de eliminación y desdoblamiento (*splitting*) ante Gaussianas "huérfanas" (componentes con un peso excesivamente reducido).

Sautrela contiene un procesador llamado `GMMTrainer` (véase Apéndice C) diseñado para llevar a cabo el entrenamiento de un GMM a partir de un conjunto de muestras. De manera análoga al entrenamiento de WFSAs (véase Sección 5.2), los GMMs disponen de una interfaz de entrenamiento. De ahí que el control de cuáles de los parámetros internos son re-estimados y cuáles no, resida en el propio GMM. En concreto, es posible establecer de manera independiente si la re-estimación afectará o no a las medias, las varianzas y los pesos, así como la posibilidad de regeneración de componentes "huérfanas".

Opciones y argumentos

-m Se desactiva el entrenamiento de medias.

-M Se activa el entrenamiento de medias [activado por defecto].

-s Se desactiva la eliminación y desdoblamiento ante Gaussianas huérfanas.

-S Se activa la eliminación y desdoblamiento ante Gaussianas huérfanas [activado por defecto]. Aquellas Gaussianas cuyo peso resulte inferior a $1/(100 \cdot g)$ (siendo g el número de componentes del GMM) son eliminadas, realizándose un desdoblamiento (*splitting*) de la Gaussiana más probable.

-v Se desactiva el entrenamiento de varianzas.

-V Se activa el entrenamiento de varianzas [activado por defecto].

-w Se desactiva el entrenamiento de pesos.

-W Se activa el entrenamiento de pesos [activado por defecto].

-n name Nombre del modelo resultante [`noName` por defecto].

-g gNum Número de Gaussianas del modelo resultante [`1` por defecto]. La reducción de Gaussianas se realiza mediante la eliminación de las Gaussianas menos probables, repartiendo su peso a partes iguales entre el resto de Gaussianas. Por el contrario, el incremento de Gaussianas se realiza mediante el desdoblamiento aleatorio (*splitting*) de la Gaussiana de mayor peso. Dicho desdoblamiento se obtiene generando un vector aleatorio de medias en torno al vector de medias original, haciendo uso de la varianza mínima. La varianza y la varianza mínima de la nueva Gaussiana son idénticas a las originales y el peso es compartido a partes iguales por ambas Gaussianas (la Gaussiana original ve reducido su peso a la mitad). En el caso concreto de utilizar el comando `GMM` para la creación de un nuevo modelo, es recomendable no generar más de una componente, ya que al no haber sido estimada aún la varianza mínima, se hará uso del valor actual de la varianza (`1.0` para las nuevas componentes).

-d gDim Dimensión de las Gaussianas del modelo resultante [`1` por defecto]. La reducción de dimensionalidad se realiza eliminando las últimas dimensiones de cada Gaussiana. El aumento de dimensionalidad se realiza añadiendo medias `0.0`, varianzas `1.0` y varianzas mínimas `0.0` ($varFloor = 0{,}0$ corresponde a una componente sin estimar). La varianza mínima es establecida en la primera estimación del GMM, donde le será asignado un valor dos órdenes de magnitud inferior a la varianza estimada ($varFloor = var_{ini}/100$).

URL Localizador de un GMM existente. El GMM cargado es modificado a partir de los parámetros especificados en la línea de comando y el modelo resultante es enviado a la salida estándar. Aquellos parámetros que no sean especificados no son modificados (los valores por defecto son efectivos únicamente si en la línea de comando no se indica el localizador de un GMM).

Ejemplos

Para generar un GMM (`ubm.1.gmm`) con una única Gaussiana inicial de dimensión 13:

```
$ sautrela GMM -n myUBM -d 13 > ubm.1g.gmm
```

Para redimensionar el GMM `ubm.1.gmm` a dos Gaussianas:

```
$ sautrela GMM -g 2 ubm.1g.gmm > ubm.2g.gmm
```

Vea también

CHMM,CHMMEdit

HTKResource

Nombre

`edu.gtts.sautrela.htk.HTKResource` - Muestra la información relativa al formato de un conjunto de recursos de datos en formato HTK.

Sinopsis

```
HTKResource URL ...
```

Descripción

Analiza un conjunto de recursos de datos en formato HTK a través de sus localizadores y muestra la información relativa a su formato (tipo de parámetros y modificadores, número y dimensión de las muestras y periodo de muestreo). El comando genera un mensaje de error cuando el recurso no es accesible o cuando el esquema del localizador URL no está soportado (véase Sección 3.7). También se genera un mensaje de error si el formato del recurso no es compatible (véase Sección 4.3). Sautrela no es compatible con recursos que hagan uso de las características de *Compresión* y *Chequeo de CRC* de HTK.

Opciones y argumentos

URL Localizador URL del recurso HTK a analizar.

Ejemplos

Para analizar todos los ficheros de datos con extensión `mfcc` del directorio actual:

```
$ sautrela HTKResource *.mfcc
Resource:  file:x001.mfcc
  ParameterKind MFCC
  Qualifiers D A Z O
  Samples 295
  SampleDim 19
  SamplePeriod 100000
Resource:  file:x002.mfcc
  ParameterKind MFCC
  Qualifiers D A Z O
  Samples 325
  SampleDim 19
  SamplePeriod 100000
```

Vea también

AcousticDataBase, AudioResource

KTLSS

Nombre

`edu.gtts.sautrela.wfsa.models.KTLSS` - Crea o modifica un modelo KTLSS.

Sinopsis

```
KTLSS [-gG] [-n name] [-N maxN] [-p minCount] [-v vocSize] [URL]
```

Descripción

Crea o carga un modelo de lenguaje K-testable en sentido estricto (KTLSS, por sus siglas en inglés) y vuelca a la salida estándar una versión modificada del mismo. Un modelo KTLSS es un autómata de estados finitos ponderado determinista que representa la gramática de un Lenguaje K-Testable en Sentido Estricto, una subclase de los Lenguajes Regulares capaz de describir la misma distribución de probabilidad que los N-gramas, permitiendo la integración eficaz del mecanismo de suavizado por back-off [174, 26, 179]. Este comando permite realizar dos tareas principales:

- **Generar un KTLSS inicial.** Se crea un KTLSS inicial (vacío) que puede posteriormente ser entrenado a partir de un conjunto de muestras.
- **Modificar un modelo ya existente.** Es posible modificar el nombre, la profundidad de n-gramas, el tamaño de vocabulario, el criterio de poda o la capacidad de modificar la topología de un KTLSS dado.

Opciones y argumentos

-g Se desactiva el comportamiento dinámico del KTLSS en tiempo de entrenamiento. Solo se re-estiman las probabilidades de las transiciones.

-G Se activa el comportamiento dinámico del KTLSS en tiempo de entrenamiento [activado por defecto]. Durante la fase de entrenamiento, la topología de un KTLSS puede crecer de forma dinámica, generando nuevas transiciones allá donde ocurran n-gramas no vistos.

-n name Nombre del autómata resultante [`noName` por defecto].

-N maxN Número máximo para N en los N-gramas a estimar [`N=3` por defecto].

-p minCount Cuenta mínima de suavizado [`0` por defecto]. Al finalizar la fase de entrenamiento, son descartados aquellos estados que tengan un número de apariciones inferior.

-v vocSize Tamaño estimado del vocabulario [`0` por defecto]. El tamaño estimado del vocabulario afecta al cálculo de probabilidades de las palabras fuera de vocabulario (OOV). Si en una fase de re-estimación el número de palabras vistas resulta superior al tamaño estimado, este es restablecido al doble del valor observado.

URL Localizador de un KTLSS existente. Si no se aporta un modelo de referencia, se crea un KTLSS vacío (contiene un único estado de *backoff* que representa el N-grama con $N = 0$) con todos los parámetros por defecto.

Ejemplos

Para generar un KTLSS inicial para trigramas, con una cuenta mínima de poda `2` y vocabulario estimado en `1000` palabras:

```
$ sautrela KTLSS -n myKTLSS -p 2 -N 3 -v 1000 > 3gram.ktlss
```

Para desactivar el comportamiento dinámico de un KTLSS, limitando así su posterior entrenamiento a la re-estimación de transiciones ya vistas:

```
$ sautrela KTLSS -g 3gram.ktlss > 3gram_static.ktlss
```

Vea también

TreeModel

LMM

Nombre

`edu.gtts.sautrela.wfsa.models.LMM` - Genera un modelo de Markov por capas (*Layered Markov Model*, LMM).

Sinopsis

```
LMM [-c] [-d dLayer] [-n name] <-e | -s | wfsaURL [a b]> <wfsasetURL
[a b]> ...
```

Descripción

Crea un modelo de Markov por capas (*Layered Markov Model*, LMM) [121] a partir de conjuntos de WFSAs y lo vuelca a la salida estándar. Un LMM integra en un único WFSA no-determinista distintos niveles de conocimiento representados por conjuntos de autómatas. Cada capa, compuesta por un conjunto de WFSAs (salvo la capa superior, que contiene un único autómata), modela sus unidades (clases) en función de las unidades de la capa inferior. Un LMM puede contener cualquier número de capas, y los conjuntos de WFSAs que componen las capas pueden contener cualquier número y tipo de modelos que implementen la interfaz WFSA de Sautrela[4]. Existen diversos escenarios de uso para los LMMs:

- **Entrenamiento de un conjunto de modelos a partir de recursos semi-supervisados.** A menudo no se cuenta con conjuntos de muestras que representen cada una de las clases que se pretende entrenar, sino que se parte de recursos con cierto grado de supervisión. Por ejemplo, es común contar con grabaciones de audio transcritas, en las cuales se tiene cierto grado de conocimiento sobre la secuencia de palabras (o fonemas) pronunciada, pero se desconoce la correspondencia temporal (segmentación) de cada una de las unidades transcritas. Un LMM de dos capas ofrece la posibilidad de definir una capa superior que represente el conocimiento que se tiene de los recursos y una capa inferior compuesta por los modelos a entrenar[5].

- **Re-entrenamiento acoplado de un conjunto de capas de conocimiento.** Normalmente, los conjuntos de modelos que componen cada una de las capas de conocimiento de un decodificador suelen ser entrenadas por separado. Sin embargo, y dado que el entrenamiento de un LMM conlleva el entrenamiento de cada una de sus capas, es posible realizar una fase de re-entrenamiento acoplado de las distintas capas de conocimiento.

[4] Los WFSAs que modelan un espacio continuo (como por ejemplo los CHMM) solo pueden ser integrados en la capa inferior de un LMM, ya que de integrarlos en una capa intermedia, y dado que su alfabeto es infinito, la capa inferior correspondiente debería contener un número infinito de modelos).

[5] El módulo de entrenamiento de Sautrela (véase el Procesador Trainer) realiza esta composición para cada una de las secuencias de entrada, entrenando un conjunto de WFSAs a partir de secuencias de entrada transcritas.

- **Decodificación basada en múltiples niveles de conocimiento.** Los decodificadores complejos suelen integrar diversos niveles de conocimiento. Por ejemplo, un reconocedor del habla continua suele contener al menos tres niveles de modelado: el acústico, el léxico y el de lenguaje. Un LMM permite integrar dichos niveles en un único WFSA no-determinista que podrá ser utilizado como decodificador.

La Sección 5.4 contiene una descripción más extensa de los LMMs.

Opciones y argumentos

-c Chequea la coherencia ascendente del conjunto de capas. En un modelo coherente, existe una correspondencia de nombres entre el alfabeto de emisión de toda capa y el conjunto de modelos de la capa inferior: para cada símbolo del alfabeto de la capa superior existe en la capa inferior un modelo de nombre coincidente, y para cada modelo de la capa inferior existe un símbolo de nombre coincidente, respectivamente. Para que un LMM sea operable, sin embargo, es suficiente con que dicha coherencia sea solo descendente (todos los símbolos del alfabeto quedan representados por modelos de la capa inferior). Un LMM con incoherencia ascendente (algún modelo de una capa no queda representado por el alfabeto de la capa superior), aun siendo operativo, está compuesto por capas que contienen modelos desechables, lo que podría ser indicio de un posible error de diseño.

-d dLayer Índice de la capa (0 para la capa inferior, $n-1$ para la capa superior) que determina los nombres asociados a las transiciones [n-1 por defecto, capa superior]. El nombre de una transición es utilizado como resultado de la decodificación, y puede no coincidir con el nombre del símbolo asociado a la transición. Dado que un LMM tiene multitud de capas de conocimiento, y cada una de ellas lleva asociado un alfabeto de símbolos, el parámetro `dLayer` determina cuál es la capa de conocimiento que asigna los nombres a las transiciones. Dada una transición t del LMM, si dicha transición lleva asociada una transición t_{dLayer} del nivel *dLayer* seleccionado, el nombre de la transición será el de la transición t_{dLayer}, y si no, será nulo:

$$name\,(t) = \begin{cases} name\,(t_{dLayer}) \ , & \exists\, t_{dLayer} \in t \\ \texttt{null} \ , & \nexists\, t_{dLayer} \in t \end{cases}$$

-n name Nombre del modelo resultante.

-e Equiprobable. La capa superior es creada automáticamente y asigna la misma probabilidad a cualquier combinatoria de la capa inferior, lo que equivale a no aportar ninguna información previa (esta configuración también es conocida como *open-loop*). La capa se implementa mediante un sencillo WFSA determinista con un solo estado y una distribución equiprobable no normalizada (se asigna la probabilidad 1 a toda posible transición). Esta opción es incompatible con la opción `-s`.

-s Selector. La capa superior es creada automáticamente y asigna la misma probabilidad a toda secuencia de longitud unidad de la capa inferior. La capa actúa

como un selector de la clase de mayor probabilidad de la capa inferior. La capa se implementa mediante un sencillo WFSA determinista con dos estados, inicial y final, y transiciones equiprobables para el conjunto de los símbolos que representa los modelos de la capa inferior. Esta opción es incompatible con la opción `-e`.

wfsaURL Localizador del WFSA que define la capa superior. Dado que todo LMM debe tener una capa superior compuesta por un único WFSA, este argumento es incompatible con el uso de las opciones `-e` o `-s` y es obligatorio en caso de no utilizar ninguna de ellas.

wfsasetURL Localizador del conjunto de WFSAs que componen una capa. Las capas son insertadas en orden descendente (primero la capa superior, después el primer conjunto de WFSAs, luego el siguiente, etc.).

a Escalado logarítmico aplicado a las probabilidades de la capa [1 por defecto]. Permite ponderar cada una de las capas respecto del resto. Un peso mayor añade importancia a los modelos de la capa (su información resultará más decisiva), mientras que un peso inferior minimiza la importancia relativa de la capa.

b Desplazamiento logarítmico aplicado a las probabilidades de la capa [0 por defecto]. Un desplazamiento negativo equivale a una penalización por inserción, mientras que un desplazamiento positivo equivale a una bonificación por inserción. El desplazamiento, al igual que el escalado, es específico de cada una de las capas. El par de parámetros **a** y **b** definen una transformación afín (un escalado y un desplazamiento) sobre los logaritmos de las probabilidades de cada capa (por separado):

$$log\,(\tilde{p}_{layer}) = a \cdot log\,(p_{layer}) + b$$

Ejemplos

Para crear un LMM que dé lugar a un sencillo decodificador acústico-fonético, partiendo de un conjunto de modelos fonéticos (fichero `phones.64g.chmmset`) sobre el cual añadir una capa superior equiprobable (es decir, no se añade información previa alguna sobre las secuencias a decodificar):

```
$ sautrela LMM -en myDecoder phones.64g.chmmset > phoneDecoder.lmm
```

para crear un LMM que dé lugar a un decodificador acústico-fonético que contenga además un modelo fonotáctico basado en 4-gramas de fonemas (fichero `phones.4gram.ktlss`):

```
$ sautrela LMM -n myDecoder phones.4gram phones.64g.chmmset \
  > phoneDecoder.lmm
```

Para generar un reconocedor de comandos basado en un conjunto de modelos fonéticos (fichero `phones.64g.chmmset`) y un conjunto de modelos léxicos (fichero

`lexicon.wfsaset`) que represente los comandos a reconocer (la capa superior será selectora, ya que únicamente estamos interesados en el comando más probable):

```
$ sautrela LMM -sn myCommands lexicon.wfsaset phones.64g.chmmset \
  > commandRec.lmm
```

Para crear un LMM que funcione como reconocedor automático del habla a partir de un conjunto de modelos fonéticos (fichero `phones.64g.chmmset`), un conjunto de modelos léxicos (fichero `lexicon.wfsaset`) y un modelo del lenguaje (fichero `words.4gram.ktlss`), aplicando una atenuación de 0.1 a los modelos acústicos y una penalización a la inserción de fonemas de 7 [6]:

```
$ sautrela LMM -n myDecoder words.4gram.ktlss lexicon.wfsaset 1 -7 \
  phones.64g.chmmset 0.1 0 > speechRec.lmm
```

Partiendo de un conjunto de modelos acústicos por lengua[7] (ficheros `langX.chmmset`) y un modelo fonotáctico de n-gramas de fonemas por lengua (ficheros `langX.ktlss`), los comandos que siguen generan un LMM que funciona como detector de la lengua hablada en una secuencia de entrada. La capa selectora del extremo superior seleccionará qué lengua (cuál de los modelos fonotácticos) es la más probable:

```
$ sautrela WFSASet -n "Multi lang phone CHMMs" lang1.chmmset \
  lang2.chmmset lang3.chmmset > multilang.chmmset
$ sautrela WFSASet -n "Multi lang phone Ngrams" lang1.ktlss lang2.ktlss \
  lang3.ktlss > multilang.ktlss.wfsaset
$ sautrela LMM -sn myLangId multilang.ktlss.wfsaset multilang.chmmset \
  > langId.lmm
```

Basándose en los mismos modelos del anterior ejemplo, se puede generar un LMM capaz de segmentar una señal de entrada de acuerdo a la lengua hablada. La capa superior permitirá cualquier secuencia de lenguas:

```
$ sautrela LMM -en myLangTrack multilang.ktlss.wfsaset \
  multilang.chmmset > langTrack.lmm
```

Vea también

WFSASet

[6] La atenuación acústica equivale a un escalado de los logaritmos de las probabilidades acústicas, mientras que una penalización a la inserción de fonemas equivale a un desplazamiento negativo aplicado a los logaritmos de las probabilidades léxicas.

[7] El presente ejemplo presupone que los conjuntos de nombres de los modelos acústicos de cada lengua son disjuntos.

ProcessorNavigator

Nombre

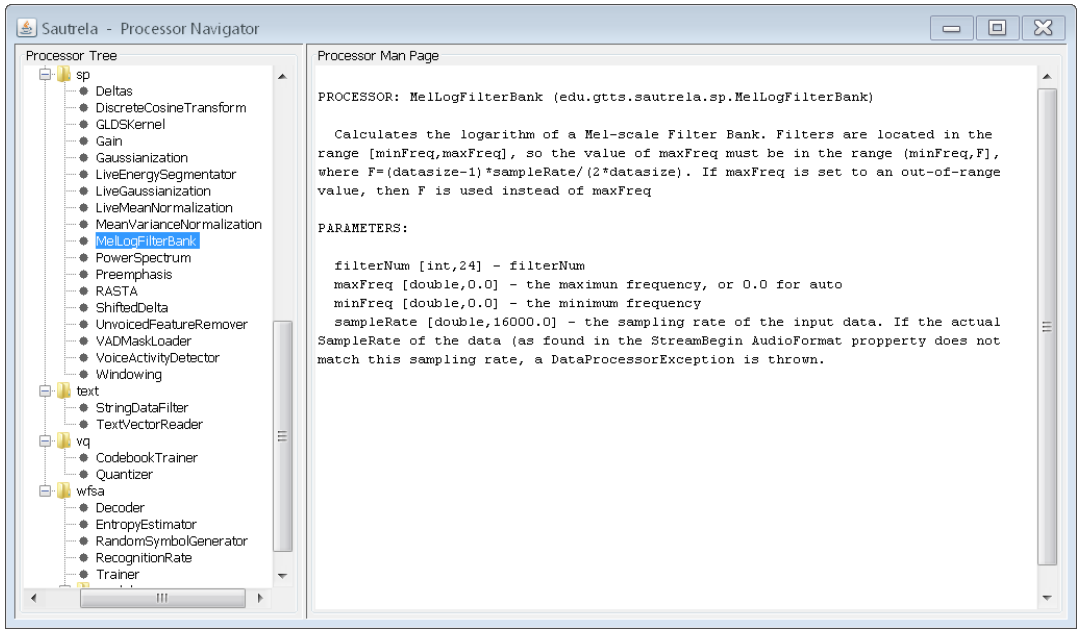
`edu.gtts.sautrela.app.ProcessorNavigator` - Aplicación gráfica que muestra el conjunto de *Procesadores* contenidos en plugins propios y de terceros.

Sinopsis

```
ProcessorNavigator
```

Descripción

Ejecuta la interfaz gráfica que permite consultar el conjunto de *Procesadores* contenidos en plugins propios y de terceros, así como su correspondiente documentación:



Vea también

CommandNavigator, Demos, EngineBuilder, ProcessorNavigator, sautrela

sautrela

Nombre

`sautrela` - Ejecuta o documenta un *Comando*, un *Procesador* o una *Engine*.

Sinopsis

```
sautrela [OPTIONS] [<Command|Processor|Engine> [args ...]]
```

Descripción

Se trata del comando principal que permite, a su vez, ejecutar o documentar un *Comando*, un *Procesador* o una *Engine*. Si la línea de comando no contiene ningún argumento y la opción `-help` ha sido activada, se muestra la página de manual del propio comando `sautrela`, y si la opción `-help` no ha sido activada, se ejecuta una aplicación gráfica que da acceso al conjunto de aplicaciones gráficas contenidas en Sautrela (véanse los comandos CommandNavigator, Demos, EngineBuilder y ProcessorNavigator). Si la línea de comando contiene al menos un argumento, se procede de la siguiente manera, en base a dicho argumento:

1. Si existe un *Comando* cuyo nombre coincide con el argumento, este es tratado como un *Comando* y, de lo contrario, se pasa a la segunda fase. Si la opción `-help` ha sido activada, se muestra la página de manual del *Comando* y se descarta el resto de argumentos. Si la opción `-help` no ha sido activada, se ejecuta el *Comando*, pasándole como argumentos el resto de los argumentos de la línea de comando.

2. Si la opción `-help` ha sido activada y existe un *Procesador* cuyo nombre coincida con el argumento, este es tratado como un *Procesador* y se muestra su página de manual (los *Procesadores* no son ejecutables y la única alternativa de línea de comando es acceder a su página de manual). De lo contrario, se pasa a la tercera fase.

3. El argumento es tratado como el localizador (URL) de una *Engine*. Si la opción `-help` ha sido activada, se muestra la página de manual de la engine. Si la opción `-help` no ha sido activada, se ejecuta la *Engine*, pasándole como argumentos el resto de los argumentos de la línea de comando.

Opciones y argumentos

-help Muestra la documentación. De no existir argumento alguno, muestra la página de manual del propio comando `sautrela`. Si existe algún argumento, muestra la página de manual del *Comando*, el *Procesador* o la *Engine* correspondiente.

-commonSeed value Modifica la semilla común de los generadores aleatorios (o `null` para desactivar el uso de una semilla común). Sautrela usa por defecto una semilla común para los generadores de números aleatorios. Ello permite que los

experimentos puedan ser reproducibles. Al modificar la semilla común, y si el procesamiento depende de algún generador aleatorio, los resultados seguirán siendo reproducibles pero probablemente diferentes a los originales. Si se asigna el valor `null` a la semilla común, la semilla de cada generador es establecida de manera automática (e impredecible) por la JVM, por lo que los experimentos dejarán de ser reproducibles.

-trace Activa el modo *desarrollo*. En caso de ocurrir alguna excepción, se muestra la traza completa de la excepción. El modo por defecto es el de *usuario*, en el cual solo se muestra una breve descripción de la excepción.

-version Muestra la versión del producto y finaliza la ejecución.

Command Nombre simple[8] o completo del *Comando*.

Processor Nombre simple[8] o completo del *Procesador*.

Engine Localizador URL de una *Engine*.

args Argumentos propios del Comando o la *Engine*.

Ejemplos

Para obtener la página de manual del Comando `Dictionary`:

```
$ sautrela -help Dictionary

  Dictionary (edu.gtts.sautrela.wfsa.models.Dictionary)

    create a Dictionary from transcriptions

  Syntax:  Dictionary  [-e enc] URL

    -e enc  The charset to use for text reading (default: "ISO8859-15")
    URL  Locator of a transcription text resource (one transcription per line)
```

Para obtener la página de manual del Procesador `Windowing`:

```
$ sautrela -help Windowing

PROCESSOR: Windowing (edu.gtts.sautrela.sp.Windowing)

  Does the windowing of the input data, aplying different transfornation
functions

PARAMETERS:

  cutDC [boolean,false] - if true, DC component is substracted for each window
```

[8] En caso de no dar lugar a ambigüedad, es posible hacer uso del nombre simple de clase.

```
  function [NONE|SQUARE|HAMMING|HAN|BLACKMAN|BLACKMAN_HARRIS,HAMMING] -
    windowing function.
  shift [int,160] - displacement of the sliding window
  size [int,400] - size of the sliding window
```

para obtener la página de manual de la Engine `ASRParam16k.eng` (véase el código del Listado 4.4):

```
$ sautrela -help ASRParam16k.eng

ENGINE: ASRParam16k

  16kHz audio parametrization for Speech Recognition:

 13MFCC + Delta-DeltaDelta + CMN

 - 25ms windows, 10ms shift, HAMMING
 - PowerSpectrum (FFTsize=512)
 - MelLogFilterBank (40 filters, minFreq=130.0Hz, maxFreq=6800.0Hz)
 - DCT (energyIncluded, filters=13)
 - Deltas
 - Mean Normalization (CMN)

PARAMETERS:

  -d [URL,file:OPENDIALOG] the locator of the AcousticDataBase or "file:
    OPENDIALOG" for a File Open Dialog [databaseURL,ADBReader]
  -s [String,".*"] the sentences whose names match this regex are read [
    resourceNameRegex,ADBReader]
  -o [File,out.dump] the pathname of the dump file [streamFile,StreamWriter]
```

Vea también

CommandNavigator, Demos, EngineBuilder, ProcessorNavigator

TreeModel

Nombre

`edu.gtts.sautrela.wfsa.models.TreeModel` - Crea un WFSA no determinista que integra un modelo de lenguaje y un léxico en árbol.

Sinopsis

```
TreeModel [-d LANGUAGE|LEXICON] [-n name] langURL dictURL
```

Descripción

Genera un WFSA no determinista que integra en un único autómata un modelo de lenguaje y un léxico en árbol, y lo vuelca a la salida estándar. Un TreeModel implementa internamente la decodificación mediante árbol léxico, ofreciendo un rendimiento eficiente en aquellos casos en los que se desea generar un decodificador que contenga un léxico de tamaño considerable. La Sección 5.7 contiene una descripción más detallada de este modelo.

Opciones y argumentos

`-d LANGUAGE|LEXICON` Determina qué capa es la responsable de generar los símbolos de decodificación [`LANGUAGE` por defecto]. Si la capa de decodificación es la de lenguaje (`-d LANGUAGE`), la decodificación dará como resultado secuencias de símbolos correspondientes al alfabeto del modelo de lenguaje (típicamente palabras). Si, por el contrario, la capa de decodificación es la del léxico (`-d LEXICON`), la decodificación dará como resultado secuencias de símbolos correspondientes al alfabeto del conjunto de modelos léxicos (típicamente fonemas).

`-n name` Nombre del WFSA resultante [`noName` por defecto].

langURL Localizador del WFSA determinista que representa el modelo de lenguaje.

dictURL Localizador del diccionario que contiene el léxico (véase el comando Dictionary). El diccionario puede contener transcripciones múltiples.

Ejemplos

Para generar un TreeModel que integre un modelo de lenguaje de 3-gramas (fichero `3gram.wfsa`) y un diccionario de 25000 palabras (fichero `25k.dict`) y que decodifique a nivel de palabra:

```
$ sautrela TreeModel -n "3gram LM + 25k vocab" 3gram.wfsa 25k.dict \
  > treeModel.wfsa
```

Vea también

DefaultDWFSA, Dictionary, KTLSS

WFSASet

Nombre

`edu.gtts.sautrela.wfsa.WFSASet` - Crea un conjunto de WFSAs.

Sinopsis

```
WFSASet [-n name] URL ...
```

Descripción

Crea un conjunto de autómatas de estados finitos ponderados (WFSAs, por sus siglas en inglés) a partir de WFSAs ya existentes y lo vuelca a la salida estándar. Se trata de un comando genérico que permite crear un conjunto a partir de cualquier tipo de modelos, siempre que cumplan con los siguientes requisitos:

- Todos los WFSAs de un conjunto deben tener nombres diferentes.
- Todos los WFSAs de un conjunto deben compartir un alfabeto único. En el proceso de instanciación de los WFSAs, el alfabeto del primer modelo instanciado es utilizado posteriormente para instanciar el resto de WFSAs. Si bien este hecho implica que el alfabeto del primer WFSA deberá dar cobertura al resto de WFSAs, depende de la implementación concreta de los WFSAs el ofrecer dicha cobertura de manera estática (el alfabeto del primer WFSA representará el alfabeto completo) o dinámica (el alfabeto del primer WFSA solo da cobertura inicial a dicho WFSA, pero es capaz de adaptarse dinámicamente a medida que se instancian nuevos WFSAs). Por ejemplo, la implementación del modelo DHMM (véase el comando DHMM) de Sautrela hace uso de un alfabeto estático, de tal manera que no es posible integrar en un mismo WFSASet distintos DHMMs que no compartan un mismo conjunto de símbolos de emisión. Por el contrario, la implementación del modelo DefaultDWFSA (véase el comando DefaultDWFSA) hace uso de un alfabeto dinámico, lo que permite que modelos que en principio no comparten un mismo conjunto de símbolos de emisión[9] puedan conformar un WFSASet.

Los WFSAs que conforman un conjunto pueden pertenecer incluso a diferentes clases, siempre que cumplan con las condiciones anteriores. Existen además comandos específicos que permiten crear o manipular conjuntos de WFSAs concretos (véanse los comandos CHMMSetEdit y DefaultDWFSASet).

Opciones y argumentos

-n name Nombre del conjunto resultante.

URL Localizador del WFSA o conjunto de WFSAs a añadir.

[9] Un modelo DefaultDWFSA puede ser utilizado, por ejemplo, como modelo de transcripción de una palabra. En tal caso, el alfabeto original de dicho modelo está formado únicamente por los símbolos o fonemas que conforman la transcripción.

Ejemplos

Para crear un WFSASet que represente un sencillo léxico de cuatro palabras:

```
$ sautrela DefaultDWFSA -n casa k a s a > casa.wfsa
$ sautrela DefaultDWFSA -n encargo e n k a r g o > encargo.wfsa
$ sautrela DefaultDWFSA -n hora o r a > hora.wfsa
$ sautrela DefaultDWFSA -n trasnochar t r a s n o c a r > trasnochar.wfsa
$ sautrela WFSASet -n myLexicon casa.wfsa encargo.wfsa hora.wfsa \
  trasnochar.wfsa > myLexicon.wsaset
```

Vea también

CHMMSet, DefaultDWFSASet

Apéndice C

Sautrela: Procesadores

El presente apéndice contiene una descripción detallada de todos los *Procesadores* incluidos en Sautrela. En la Figura C.1 se muestra la estructura de la información aportada para cada procesador. El apartado denominado "Características" describe el comportamiento de un procesador en base a las siguientes cuatro propiedades:

Tipo: `Generador | Transformador | Analizador`. Se denomina **Generador** al procesador que es encargado de generar (o cargar desde algún recurso) las secuencias de datos que vuelca a la salida. Un Generador siempre debe estar al inicio de una cadena de procesamiento ya que, aunque es posible colocarlo en una fase intermedia, ignora el buffer de entrada y, por ende, todas las fases previas de procesamiento. Se denomina **Transformador** al procesador cuya finalidad última es la transformación de la información proveniente del buffer de entrada, para volcarla al buffer de salida. La transformación de la información de entrada puede no solo modificar el contenido de cada dato de entrada, sino modificar el tipo de dato o la estructura misma de los flujos de entrada (modificar la longitud de las secuencias, filtrar secuencias completas, combinarlas, etc.). Se denomina **Analizador** al procesador que analiza/procesa los datos de entrada y los vuelca al buffer de salida sin modificación alguna. El resultado del análisis pudiera ser mostrado al usuario, ser almacenado en algún recurso o ser añadido a la cabecera de cada secuencia analizada[1]. Transformadores y Analizadores pueden ser colocados en cualquier fase de la cadena de procesamiento a excepción de la primera, donde siempre deberá colocarse un Generador.

Entrada: Tipo de flujo de datos de entrada. Algunos procesadores admiten secuencias de datos de entrada de un único tipo (`IntData`, `DoubleData` o `StringData`), mientras que otros admiten varios o todos los tipos de datos. En el caso de los datos numéricos (`IntData` y `DoubleData`), y dado que cada dato contiene un vector de valores, los valores de dicho vector pueden ser interpretados bien como una secuencia temporal de datos unidimensionales (*flujo escalar*) o bien como una

[1] Aquellos procesadores que solo modifican la cabecera de las secuencias no son considerados `Transformadores`, sino `Analizadores`.

única observación multidimensional (*flujo vectorial*). Los flujos vectoriales normalmente están compuestos por secuencias de vectores que mantienen el tamaño (la dimensión), mientras que las secuencias escalares pueden contener vectores de diferente tamaño. La adquisición de audio, por ejemplo, suele requerir la lectura periódica del buffer de audio, no resultando ni predecible ni constante el número de muestras disponibles en cada lectura.

Salida: Tipo de flujo de datos de salida. Caracteriza el flujo de salida de manera análoga a lo dispuesto con la entrada.

Retardo: Retardo añadido por el procesador. Algunos procesadores realizan un *buffering* parcial o total de la secuencia de entrada, lo que conlleva retardos en la salida. En aquellos casos en los que el retardo de la cadena de procesamiento completa no deba superar cierta magnitud, deberá tenerse en cuenta el retardo introducido por cada uno de los procesadores. El procesamiento de señales en directo (*live input*), por ejemplo, puede no ser compatible con procesadores que añadan mucho retardo. En aquellos casos en los que el retardo sea mínimo (como por ejemplo, el cálculo de derivadas), se entiende que es despreciable.

Una versión reducida de la información contenida en este apéndice para cada procesador puede obtenerse directamente del comando `sautrela`[2] mediante la opción `-help` y su nombre. El nombre, como es norma general, deberá ser completo o podrá ser sencillo, en función de que sea ambiguo o no en el conjunto de paquetes disponible:

```
$ sautrela -help MyProcessor

PROCESSOR: MyProcessor (path.to.mypackage.MyProcessor)

  This is the documentation of MyProcessor

PARAMETERS:

  param1 [type,defaultValue] - description of param1
  param2 [type,defaultValue] - description of param2
  ...
```

Durante la ejecución de una engine, puede producirse un error en alguno de los procesadores. En tal caso, concluirá la ejecución de la engine y se volcará a la salida de error un mensaje informativo detallado de lo sucedido.

[2]El comando `sautrela` no es otra cosa que un alias de: `<path_to_java_command> -jar <path_to_Sautrela.jar>` . El Apéndice A contiene una explicación más detallada sobre la instalación de Sautrela en diferentes sistemas operativos.

MyProcessor

Nombre

`path.to.mypackage.MyProcessor` - Descripción breve del procesador.

Descripción

Descripción pormenorizada del procesador.

Características

Tipo `Generador | Transformador | Analizador`.

Entrada Descripción del flujo de datos de entrada admitido.

Salida Descripción del flujo de datos de salida generado.

Retardo Descripción del retardo añadido por el procesador.

Parámetros

- **nombre1** `[tipo,valor por defecto]` Descripción del parámetro.
- **nombre2** `[tipo,valor por defecto]` Descripción del parámetro.
- ...

Vea también

Lista de procesadores relacionados.

Figura C.1 Ejemplo ilustrativo de la página de manual de un procesador de Sautrela.

ADBReader

Nombre

`edu.gtts.sautrela.db.ADBReader` - Carga un subconjunto de recursos desde una base de datos acústica (ADB).

Descripción

Carga desde una base de datos acústica de Sautrela (véase Sección 4.4) el subconjunto de recursos definido bien mediante un fichero índice, bien mediante una expresión regular. Una base de datos acústica puede contener tanto recursos de audio como recursos de datos en formato HTK. Tanto las propiedades definidas en el descriptor XML de la base de datos (véase el ejemplo del Listado 4.3) como las relativas al formato del recurso son almacenadas en la cabecera de cada secuencia, para que los procesadores sucesivos puedan disponer de ella. En concreto, la cabecera de cada secuencia de datos contendrá el siguiente conjunto de propiedades relativas al formato del recurso en cuestión:

- Propiedades comunes:
  - `ResourceName [String]` : Valor del atributo `name` del elemento XML `Resource`.
  - `ResourceType [enum]` : Tipo de recurso (`AUDIO`, `HTK`).
  - `ResourceURL [URL]` : Valor del atributo `url` de los elementos XML `Audio` o `Htk`.
- Propiedades específicas de los recursos de Audio:
  - `AudioFileFormat [AudioFileFormat]` : Formato de fichero de audio del recurso cargado. Contiene diversa información relativa al recurso de audio, como el formato de fichero (AIFF, AU, WAVE, etc.), el formato de audio (frecuencia de muestreo, tamaño de muestra, codificación, número de canales, etc.) o propiedades opcionales asociadas al recurso (duración, autor, título, fecha, copyright, etc.)
  - `AudioFormat [AudioFormat]` : Formato de audio del recurso volcado. Para facilitar el manejo de formatos de audio hetereogéneos, Sautrela realiza una conversión automática de formato, manteniendo únicamente la frecuencia de muestreo del recurso original (véase Sección 4.2.1).
- Propiedades específicas de los recursos de datos en formato HTK:
  - `Samples [int]` : Número de muestras.
  - `SamplePeriod [int]` : Periodo de muestreo (en unidades de 100ns).
  - `SampleDim [int]` : Dimensionalidad de las muestras.
  - `ParameterKind [enum]` : Tipo de parámetros HTK (`WAVEFORM`, `LPC`, `LPREFC`, `LPCEPSTRA`, `LPDELCEP`, `IREFC`, `MFCC`, `FBANK`, `MELSPEC`, `USER`, `DISCRETE`).

  - `Qualifiers[int mask]` : Máscara de Cualificadores HTK (`E, N, D, A, C, Z, K, O, V, T`).

- Propiedades definidas por el usuario en el descriptor XML.

Características

Tipo Generador.

Entrada —

Salida Flujo escalar/vectorial de datos enteros/reales.

Retardo —

Parámetros

- **databaseURL** `[URL,file:OPENDIALOG]` Localizador de la base de datos acústica, o "`file:OPENDIALOG`" para seleccionar el recurso mediante un diálogo (interfaz gráfica o GUI). La base de datos acústica puede contener tanto recursos de audio como recursos de datos en formato HTK[3]. Los recursos de audio generan un flujo escalar de datos enteros (las muestras de audio), mientras que los recursos de datos en formato HTK generan un flujo vectorial de datos reales (los vectores de datos que han sido codificados en base al formato HTK).

- **indexURL** `[URL,null]` Localizador de un índice, o "`file:OPENDIALOG`" para seleccionarlo mediante un diálogo (interfaz gráfica). El fichero índice contendrá los identificadores de los recursos a cargar (un identificador por línea), generándose un error si alguno de los identificadores no está contenido en la base de datos. Los recursos son cargados en el orden establecido por el fichero índice.

- **resourceNameRegex** `[String,.*]` Expresión regular que describe el patrón de búsqueda. Si no se suministra ningún índice, el descriptor XML de la base de datos es recorrido secuencialmente y son cargados todos aquellos recursos cuyos identificadores se ajusten al patrón indicado. Si, por el contrario, se suministra un índice (`indexURL` $\neq$ `null`), el valor de esta propiedad es ignorado y únicamente son cargados los recursos indexados.

Vea también

AudioRecorder, AudioResourceReader, AudioPlayer

[3] Aunque una base de datos acústica puede contener ambos tipos de recurso, y por tanto es posible generar secuencias hetereogéneas de datos, es de esperar que el subconjunto de recursos seleccionado corresponda únicamente a uno de los dos tipos.

AudioPlayer

Nombre

`edu.gtts.sautrela.audio.AudioPlayer` - Reproducción de audio.

Descripción

Reproduce la señal de entrada utilizando el dispositivo de reproducción de audio por defecto del sistema. Hace uso de la propiedad `AudioFormat` que debe contener la cabecera de cada secuencia de datos para configurar el dispositivo de reproducción. En caso de no contener dicha propiedad, hace uso del formato de audio por defecto de Sautrela:

- Frecuencia de muestreo: 16000Hz
- Tamaño de muestra: 16 bits. Los datos de entrada son truncados al rango $[-2^{15}, 2^{15} - 1]$ y convertidos en muestras de audio de 16 bits.
- Codificación: lineal PCM con signo
- Canales: 1 (mono)
- Esquema multibyte: *big-endian*

Características

Tipo Analizador.

Entrada Flujo escalar de datos enteros/reales.

Salida Flujo escalar de datos enteros/reales.

Retardo La velocidad de procesamiento viene determinada por la frecuencia de muestreo de reproducción.

Parámetros

- **beepAdded** `[boolean,false]` Añade una señal sonora entre cada segmento de audio, permitiendo distinguir los límites de cada flujo de audio.

Vea también

ADBReader, AudioRecorder, AudioResourceReader

AudioRecorder

Nombre

`edu.gtts.sautrela.audio.AudioRecorder` - Captura de audio en directo.

Descripción

Realiza una captura de audio en directo, utilizando el dispositivo de captura de audio por defecto del sistema. Hace uso del formato de audio por defecto de Sautrela:

- Frecuencia de muestreo: 16000Hz
- Tamaño de muestra: 16 bits
- Codificación: lineal PCM con signo
- Canales: 1 (mono)
- Esquema multibyte: *big-endian*

siendo configurables tanto la frecuencia de muestreo como el tamaño de muestra. Este último determinará el rango de valores de las muestras ($[-2^{15}, 2^{15}-1]$ para el formato de audio por defecto, que establece muestras de 16 bits).

El procesador ofrece controles visuales para iniciar, pausar y finalizar la captura de audio. Cada vez que se reanuda la captura de audio se genera una nueva secuencia de datos, añadiéndose a la cabecera la propiedad `AudioFormat` que contendrá el formato de audio establecido.

Características

Tipo Generador.

Entrada —

Salida Flujo escalar de datos enteros.

Retardo La velocidad de procesamiento viene determinada por la frecuencia de muestreo de adquisición.

Parámetros

- **sampleFreq** `[int,16000]` Frecuencia de muestreo, en hercios.
- **sampleSize** `[int,16]` Tamaño de la muestra en bits.

Vea también

AudioPlayer

AudioResourceReader

Nombre

`edu.gtts.sautrela.audio.AudioResourceReader` - Carga un recurso de audio.

Descripción

Carga un recurso de audio a partir de su localizador, generando una única secuencia de datos. Añade a la cabecera las siguientes propiedades:

- `ResourceURL [URL]` : Localizador del recurso.
- `AudioFileFormat [AudioFileFormat]` : Formato de fichero de audio del recurso cargado. Contiene diversa información relativa al recurso de audio, como el formato de fichero (AIFF, AU, WAVE, etc.), el formato de audio (frecuencia de muestreo, tamaño de muestra, codificación, número de canales, etc.) o propiedades opcionales asociadas al recurso (duración, autor, título, fecha, copyright, etc.)
- `AudioFormat [AudioFormat]` : Formato de audio del recurso volcado. Para facilitar el manejo de formatos de audio hetereogéneos, Sautrela realiza una conversión automática de formato, manteniendo únicamente la frecuencia de muestreo del recurso original (véase Sección 4.2.1).

Características

Tipo Generador.

Entrada —

Salida Flujo escalar de datos enteros.

Retardo —

Parámetros

- **audioURL** `[URL,file:OPENDIALOG]` Localizador del recurso de audio, o "`file:OPENDIALOG`" para seleccionar el recurso mediante un diálogo (interfaz gráfica).

Vea también

ADBReader,AudioRecorder,HTKResourceReader

CodebookTrainer

Nombre

`edu.gtts.sautrela.vq.CodebookTrainer` - Estimación de un conjunto de centroides.

Descripción

Estima un codebook (conjunto de centroides) a partir de las muestras de entrada. Toma todas las muestras de la entrada, las almacena en memoria[4] y a continuación busca el conjunto de centroides que minimice la distorsión global (típicamente, la suma de los cuadrados de las distancias al centroide más próximo). La re-estimación del codebook puede realizarse en base a los algoritmos LBG [104] y eLBG *(enhanced LBG)* [117], partiendo de un conjunto vacío o un codebook previamente estimado. El algoritmo eLBG trata de evitar caer en soluciones de mínimo local a las que fácilmente tiende el tradicional algoritmo LBG. Para ello, se siguen los siguientes pasos:

1. Escoger de manera aleatoria (inversamente proporcional a su distorsión) un clúster cuya distorsión sea inferior a la media. Llamaremos a este clúster *débil*, ya que, a priori, debería ser un clúster de poca relevancia.

2. Escoger de manera aleatoria (proporcionalmente a su distorsión), otro clúster. Llamaremos a este clúster *fuerte*, ya que, a priori, debería ser un clúster de mucha relevancia.

3. Localizar el clúster más cercano al débil (aquel diferente del fuerte y cuyo centroide sea el más cercano al centroide del débil). Llamaremos a este clúster *cercano*.

4. Tomar las muestras de los tres clústers (débil, fuerte y cercano), y un codebook formado por: una muestra aleatoria del clúster fuerte, su muestra simétrica (frente al centroide del clúster) y el centroide del clúster cercano.

5. Realizar una única iteración del algoritmo LBG. Si la distorsión del nuevo codebook es inferior a la de partida (débil + fuerte + cercano), reemplazar los tres clústers.

La Figura C.2 muestra el resultado del entrenamiento de un codebook de 16 centroides dada una distribución Cantor de tres niveles [48]. Como puede observarse, el algoritmo eLBG es capaz de obtener la solución de mínima distorsión global, mientras que el algoritmo LBG tiende a converger a soluciones de mínimo local.

Características

Tipo Analizador.

[4] Dado que este procesador trabaja sobre la totalidad de los datos una vez cargados estos en memoria, debe tenerse en cuenta que el tamaño del conjunto de datos queda limitado por la disponibilidad de memoria del sistema.

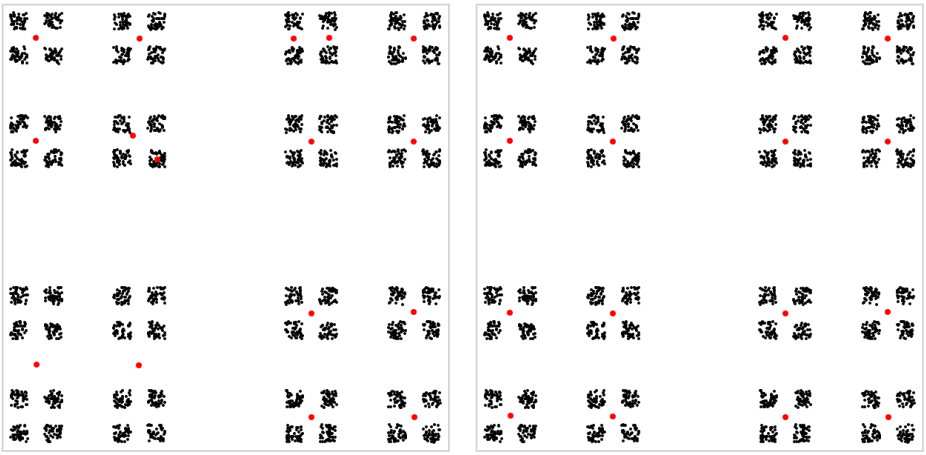


Figura C.2 Resultado del entrenamiento de 16 centroides dada una distribución Cantor de tres niveles. En negro las muestras de la distribución, y en rojo los centroides estimados. El algoritmo LBG (izquierda) cae fácilmente en soluciones de mínimo coste local. Por el contrario, el algoritmo eLBG tiene una mayor probabilidad de obtener la solución de mínima distorsión global, como sucede en este caso.

Entrada Flujo vectorial de datos reales.

Salida Flujo vectorial de datos reales.

Retardo —

Parámetros

- **enhancedLBG** `[boolean,true]` Determina la utilización del algoritmo eLBG para la estimación de centroides. De lo contrario, se utiliza el algoritmo LBG estándar.
- **binaryCdbk** `[boolean,false]` Determina si el formato del fichero de codebook es binario (`true`) o XML (`false`). Este parámetro afecta simultáneamente al recurso que contiene el codebook inicial (opcional) y al fichero donde se volcará el codebook estimado.
- **cdbkSize** `[int,10]` Número final de centroides (tamaño del codebook).
- **delta** `[double,1.0E-10]` Parámetro de convergencia. El algoritmo concluye si la reducción relativa de la distorsión global (suma de los cuadrados de las distancias al centroide más próximo) es menor que dicho parámetro.
- **finalCdbk** `[File,out.cdbk]` Ruta del fichero donde volcar el codebook estimado. Su formato dependerá del parámetro `binaryCdbk`, pero en todo caso el fichero resultante contendrá un flujo vectorial de datos reales en formato Sautrela (véase el Procesador StreamWriter), donde cada vector representará un centroide del codebook resultante.

- **incremental** `[boolean,false]` Determina si, en caso de tener que ampliar el tamaño del codebook, dicha ampliación se realiza o no de manera incremental. Ambos modos se basan en la generación aleatoria[5] de nuevos centroides y la posterior re-estimación del conjunto de centroides. Dado un codebook de tamaño actual $currentSize$ y tamaño final $cdbkSize$, ambos modos se diferencian en:
  - Modo no incremental: se generan $cdbkSize - currentSize$ centroides y se re-estima el conjunto resultante.
  - Modo incremental: se aumenta el tamaño del codebook, generando $\text{mín}\left\{\text{máx}\left\{1, 2 \cdot currentSize\right\}, cdbkSize\right\} - currentSize$ nuevos centroides y se re-estima el conjunto resultante. Si el tamaño resultante es inferior a $cdbkSize$, se vuelve a aplicar el mismo procedimiento, aumentando y re-estimado el codebook de manera iterativa.
- **initCdbk** `[URL,null]` Localizador del recurso de datos que contiene el codebook inicial, o "`file:OPENDIALOG`" para seleccionar el recurso mediante un diálogo (interfaz gráfica). Su formato dependerá del parámetro `binaryCdbk`, pero en todo caso el recurso de datos debe contener un flujo vectorial de datos reales en formato Sautrela (véase el Procesador StreamWriter), donde cada vector representa un centroide del codebook inicial. Si el tamaño del codebook inicial es inferior al tamaño deseado (parámetro `cdbkSize`), su tamaño será ampliado hasta lograr el tamaño deseado en base al criterio del parámetro `incremental`. Si por el contrario, el tamaño del codebook inicial es superior al tamaño deseado, el procesador no modificará el codebook cargado. El valor por defecto (`null`) corresponde a partir de un codebook inicial vacío (no se carga ningún recurso de datos).
- **maxIter** `[int,100]` Número máximo de iteraciones. Limita el número máximo de iteraciones de re-estimación en caso de no llegar a converger. Si el entrenamiento incremental está activado (parámetro `incremental`), el número máximo de iteraciones es aplicado a cada fase de ampliación/re-estimación
- **verbose** `[boolean,false]` Determina si el procesador muestra o no por la salida estándar el desarrollo de las diferentes fases de la estimación del codebook.

Vea también

Quantizer

[5] La generación aleatoria de muestras se realiza mediante la selección aleatoria de muestras contenidas en el conjunto de aprendizaje.

CrossEntropyEstimator

Nombre

`edu.gtts.sautrela.wfsa.CrossEntropyEstimator` - Estimador de entropía cruzada empírica.

Descripción

Dado un autómata de estados finitos ponderado (WFSA, por sus siglas en inglés) cargado desde un recurso, realiza la estimación de la entropía cruzada empírica [196] para el conjunto de secuencias de datos de entrada. La entropía cruzada entre dos distribuciones discretas p y q sobre un conjunto de eventos X viene dada por:

$$H(p,q) = \mathrm{E}_p[-\log_2 q] = -\sum_{x \in X} p(x) \log_2 q(x)$$

La entropía cruzada mide la cantidad promedio de bits que se precisan para codificar un evento x generado de acuerdo a la distribución de probabilidad p, en base al mero conocimiento de la distribución q. Su valor está acotado inferiormente por la entropía $H(p)$ de la distribución p, ya que:

$$H(p,q) \geq H(p,p) = H(p) = -\sum_{x \in X} p(x) \log_2 p(x)$$

En otras palabras, la entropía cruzada es mínima si y solo si la distribución q coincide con la distribución real p.

En un caso concreto de aplicación, p podría representar la distribución real del conjunto de eventos X, y q representaría la distribución de probabilidad de un modelo probabilístico definido por un conjunto de parámetros θ. De esta manera, la entropía cruzada estaría indicando el grado de ajuste del modelo: cuanto "mejor" sea el modelo, menor será la entropía cruzada. En un caso real, no obstante, difícilmente conoceremos la distribución real p. En tales casos es posible realizar una estimación de la entropía cruzada en base a un conjunto T de muestras de test, también denominada *entropía cruzada empírica*:

$$H(T,q) = \frac{-1}{|T|} \sum_{i=1}^{|T|} \log_2 q(x_i|\theta)$$

Esta magnitud es por tanto una estimación empírica de la cantidad promedio de bits necesarios para predecir o codificar un símbolo de entrada dado un modelo probabilístico, y está directamente relacionada con la verosimilitud $\mathcal{L}(\theta|T)$ y la perplejidad empírica $PP(T,q)$ [205]:

$$\mathcal{L}(\theta|T) = q(T|\theta) = \prod_{i=1}^{|T|} q(x_i|\theta) \;\rightarrow\; H(T,q) = \frac{-\log_2 \mathcal{L}(\theta|T)}{|T|}$$

$$PP\,(T,q) = 2^{\frac{-1}{|T|}\sum_{i=1}^{|T|}\log_2 q(x_i|\theta)} = 2^{H(T,q)}$$

El Procesador `CrossEntropyEstimator` es, en esencia, muy similar al procesador Decoder. A diferencia de este último, se comporta como un simple analizador de datos, ya que no modifica el flujo de datos de entrada y muestra el resultado final del análisis por la salida estándar.

Características

Tipo Analizador.

Entrada Flujo genérico de datos.

Salida Flujo genérico de datos.

Retardo —

Parámetros

- **beam** `[double,-1.0]` Ancho de haz para la búsqueda de los posibles caminos, o un valor negativo para desactivar la búsqueda en haz. En cada instante (para cada muestra de entrada) se descartan todos aquellos caminos cuyos logaritmos de probabilidad disten del camino más probable una cantidad superior al valor del haz.
- **modelURL** `[URL,file:OPENDIALOG]` Localizador del recurso que contiene el WFSA a utilizar en la decodificación, o `"file:OPENDIALOG"` para seleccionar el recurso mediante un diálogo (interfaz gráfica).

Vea también

Decoder, Trainer

DataJoiner

Nombre

`edu.gtts.sautrela.engine.util.DataJoiner` - Agrupador de datos.

Descripción

Agrupa todos los datos de cada secuencia, generando secuencias que contienen un único dato:

- Cada secuencia debe contener un único tipo de dato (entero, real o de texto), de lo contrario se genera un error.
- Los datos numéricos son agrupados en un único dato numérico cuyo vector es la concatenación de todos los vectores originales.
- Los datos de texto son agrupados en un único dato de texto cuyo valor es la concatenación de todas las cadenas de texto originales. La concatenación de cadenas de caracteres hace uso del delimitador especificado.

Características

Tipo Transformador.

Entrada Flujo genérico de datos.

Salida Flujo genérico de datos.

Retardo Secuencia de datos completa.

Parámetros

- **stringJoiner** `[String," "]` Cadena utilizada como delimitador en la concatenación de cadenas de caracteres.

Vea también

VectorialDataFilter, StreamMixer

DataPlotter

Nombre

`edu.gtts.sautrela.engine.util.DataPlotter` - Representación 2D de datos.

Descripción

Obtiene una sencilla representación 2D a partir de los datos vectoriales de entrada. La representación hace uso a lo sumo de dos de las dimensiones del vector de entrada[6]. Cada secuencia de datos de entrada es representada con un color diferente, y en caso de superar la paleta de colores, el tamaño del trazo es aumentado para distinguir unas secuencias de otras. Las secuencias de datos pueden ser representadas como secuencias temporales, en cuyo caso los puntos consecutivos son unidos por puntos, o como datos independientes.

La Figura C.3 muestra dos ejemplos de uso del Procesador `DataPlotter` en las que las secuencias de datos son representadas como secuencia temporal y como datos independientes, respectivamente.

Características

Tipo Analizador.

Entrada Flujo genérico de datos.

Salida Flujo genérico de datos.

Retardo ——

Parámetros

- **height** `[int,400]` Altura en píxeles del marco o ventana gráfica.
- **style** `[DOT|LINE,LINE]` Estilo de línea a utilizar. El estilo `DOT` representa cada dato como un punto independiente, mientras que el estilo `LINE` representa la secuencia de datos como una secuencia de puntos unida por líneas.
- **width**`[int,600]` Anchura en píxeles del marco o ventana gráfica.
- **xdataIndex**`[int,-1]` Indice de la dimensión que contiene los valores a representar en el eje X, o -1 para generación automática de valores en base a la progresión aritmética: 0, 1, 2, 3, 4, ...
- **ydataIndex**`[int,0]` Indice de la dimensión que contiene los valores a representar en el eje Y, o -1 para generación automática de valores en base a la progresión aritmética: 0, 1, 2, 3, 4, ...

[6] El Procesador `DataPlotter` no realiza una proyección 2D de los vectores multidimensionales de la entrada, simplemente se limita a mostrar gráficamente los valores de, a lo sumo, dos de las dimensiones del vector de entrada.

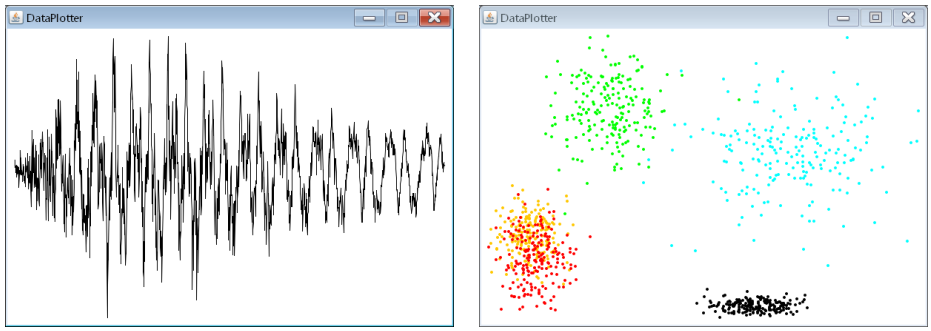


Figura C.3 Imágenes 2D generadas por el procesador `DataPlotter`. El Procesador permite representar los datos como secuencias temporales (los puntos consecutivos son unidos por puntos) o como datos independientes. Cada secuencia de datos de entrada es representada con un color diferente.

Vea también

Sniffer, StreamTester

Decoder

Nombre

`edu.gtts.sautrela.wfsa.Decoder` - Decodificador genérico de WFSAs.

Descripción

Dado un autómata de estados finitos ponderado (WFSA, por sus siglas en inglés) obtiene la secuencia más probable de transiciones (camino óptimo) para cada secuencia de datos de entrada (secuencia de observables), y vuelca a la salida la secuencia de nombres correspondiente (un nombre por cada transición).

La decodificación puede hacer uso de una búsqueda en haz (*beam*), descartando para cada muestra de entrada caminos que resulten poco probables. En caso de una configuración de haz excesivamente restrictiva que provoque que no se obtenga ningún posible camino para una secuencia de entrada, el procesamiento de la secuencia es reiniciado, aumentando el haz de búsqueda. También es posible activar el mecanismo denominado *Automatic Beam Tuning*, que estimará el valor de haz óptimo para mantener un número promedio de estados activos durante el proceso de decodificación.

El Procesador `Decoder` implementa, independientemente de la naturaleza interna de los modelos utilizados, un mecanismo unificado de decodificación capaz de obtener la secuencia más probable de transiciones para todo modelo que implemente la interfaz de decodificación de Sautrela. Las Secciones 5.1, 5.4 y 5.5 contienen una descripción más extensa del proceso unificado de decodificación y desarrollan en profundidad el formalismo de los LMMs (Layered Markov Model), gracias a los cuales es posible integrar diferentes niveles de conocimiento en un único autómata, y crear complejos sistemas de decodificación.

Características

Tipo Transformador.

Entrada Flujo genérico de datos.

Salida Flujo de datos de texto.

Retardo Secuencia completa de datos.

Parámetros

- **averageTrellisSize** `[int,0]` Número promedio de estados activos deseado. Un valor superior a 0 activa el mecanismo de ajuste automático de ancho de haz (*Automatic Beam Tuning*). Este mecanismo hace uso de las primeras secuencias de entrada (véase parámetro `automaticBeamTuningStreams`) para ajustar el valor del ancho de haz, de tal manera que el número promedio de estados activos en cada instante sea el deseado. Este mecanismo es especialmente útil cuando

se aplican transformaciones afines a los logaritmos de las distribuciones de probabilidad de los modelos (véase Comando LMM), ya que ello modifica su rango dinámico (un mismo ancho de haz puede resultar excesivamente elevado en unos casos y excesivamente reducido en otros).

- **automaticBeamTuningStreams** `[int,1]` Número de secuencias iniciales utilizadas para el ajuste automático de ancho de haz (véase parámetro `averageTrellisSize`). En ancho de haz que se utilizará para el resto de secuencias será el promedio de los valores obtenidos co cada una de las secuencias iniciales.

- **beam** `[double,-1.0]` Ancho de haz para la búsqueda del camino óptimo, o un valor negativo para desactivar la búsqueda en haz. En cada instante (para cada muestra de entrada) se descartan todos aquellos caminos cuyos logaritmos de probabilidad disten del camino más probable una cantidad superior al valor del haz.

- **beamRetryFactor** `[double,1.2]` Si el procesamiento de una secuencia de datos no da lugar a ningún camino posible (ninguno de los estados supervivientes es final), el procesamiento es reiniciado haciendo uso de un haz aumentado: $newBeam = beamRetryFactor \cdot oldBeam$.

- **beamRetryMax** `[int,15]` Número máximo de veces que se reintentará el procesamiento de la entrada con un haz aumentado. El hecho de no encontrar ningún camino posible, después de múltiples reintentos, puede indicar la existencia de algún error bien en los datos, bien en los modelos.

- **gcMinInterval** `[long,1000]` Determina el tiempo mínimo (en milisegundos) entre dos activaciones consecutivas del proceso de recolección de basura (véase parámetro `gcPolicy`).

- **gcPolicy** `[NONE|GC|FULLGC,NONE]` Determina la política de activación de recolección de basura implementada. La recolección de basura (del inglés *garbage collection*) es un mecanismo de gestión de memoria implementado por la JVM. Este mecanismo funciona de manera autónoma y normalmente no resulta preciso ni recomendable interaccionar con él, ya que se activa automáticamente, bien de manera periódica, bien cuando detecta que los recursos de memoria empiezan a ser escasos[7]. Sin embargo, en situaciones en las que el uso de memoria es elevado y predecible, puede resultar conveniente la activación del mecanismo de recolección en instantes concretos. Tal es el caso de la decodificación, si se hace uso de un modelo complejo (como por ejemplo un reconocedor de habla continua con gran vocabulario), donde el procesamiento de cada secuencia de entrada puede requerir una gran cantidad de memoria que puede ser liberada una vez finalizada la decodificación. La activación manual de la recolección de basura al finalizar el procesamiento de cada secuencia de entrada, en comparación con su activación

[7] Java implementa distintos mecanismos de recolección de basura, y lo aquí expuesto es una simplificación que en determinados casos puede resultar insuficiente, por lo que emplazamos al lector a obtener una información más detallada en [6].

automática, puede dar lugar a una mejor gestión de la memoria por parte de la JVM[8]. Las políticas de activación implementadas son:

- `NONE`. No se aplica política alguna. La recolección de basura es guiada de manera automática por la propia JVM.
- `GC`. Al finalizar el procesamiento de cada secuencia de entrada, y si ha transcurrido el tiempo mínimo establecido desde la última llamada (véase parámetro `gcMinInterval`), se realiza una llamada al método de recolección de basura.
- `FULLGC`. Al finalizar el procesamiento de cada secuencia de entrada, y si ha transcurrido el tiempo mínimo establecido desde la última llamada (véase parámetro gcMinInterval), se ejecuta un ciclo de llamadas consecutivas al método de recolección de basura hasta constatar que no se obtiene liberación alguna de espacio.

- **modelURL** `[URL,file:OPENDIALOG]` Localizador del recurso que contiene el WFSA a utilizar en la decodificación, o "`file:OPENDIALOG`" para seleccionar el recurso mediante un diálogo (interfaz gráfica).
- **verbose** `[int,0]` Nivel de verbosidad utilizado para mostrar por la salida estándar información sobre desarrollo de las diferentes fases de la decodificación. Los niveles de verbosidad definidos son:

  0 No muestra nada

  1 Muestra en una sola línea la evolución temporal de la decodificación (el texto que representa el camino óptimo). El valor final corresponde a la secuencia que será volcada por el procesador a la salida. Los niveles de verbosidad superiores no muestran esta información

  2 Muestra la cabecera de cada secuencia de datos de entrada y el tamaño promedio de trellis (número promedio de estados activos).

  3 Para cada instante (dato de entrada), muestra el tamaño del trellis (número de estados activos).

  4 Para cada instante (dato de entrada), muestra cada uno de los estados activos.

Vea también

Trainer

[8] El rendimiento de las presentes políticas de activación del proceso de recolección de basura dependerá de la configuración de la JVM y el equipo físico en el que se ejecuta, pudiendo llegar a ser contraproducente en algunos casos. Este mecanismo está desactivado por defecto y debiera ser usado en aquellos casos en los que se detecten problemas de gestión de memoria por parte de la JVM. Nótese también que estas políticas de activación son complementarias a la configuración del mecanismo de recolección de basura descritas en [6].

Delta

Nombre

`edu.gtts.sautrela.sp.Delta` - Primeras y segundas derivadas.

Descripción

Obtiene la primera y segunda derivada de cada vector de entrada. La derivada $\Delta_t(x)$ de los parámetros de entrada x en el instante t es calculada mediante la expresión [227]:

$$\Delta_t(x) = \frac{\sum_{d=1}^{D} d \cdot (x_{t+d} - x_{t-d})}{2\sum_{d=1}^{D} d^2}$$

donde D es el orden de la derivada (la ventana utilizada para su cálculo). Los vectores de salida contienen los valores estáticos, su primera y su segunda derivada (la dimensión de los vectores de salida es tres veces la dimensión de los vectores de la entrada).

La inclusión de información dinámica en los vectores de parámetros ofrece mejoras de modelado temporal en aquellos casos en los que el modelo en cuestión asuma la independencia estadística de las muestras (tal es el caso de los GMMs o modelos más complejos basados en GMMs, como los HMM continuos). En estos casos, la carencia de modelado temporal es compensada con la información temporal implícita en los parámetros. Por otro lado, y dado que los vectores de salida contendrán siempre tanto los coeficientes estáticos como los dinámicos (primera y segunda derivada), en caso de desear obtener solamente un subconjunto de ellos, este procesador debería ser seguido por el Procesador VectorialDataFilter que permite filtrar los campos deseados.

Características

Tipo Transformador.

Entrada Flujo vectorial de datos reales.

Salida Flujo vectorial de datos reales.

Retardo —

Parámetros

- **deltaDeltaOrder** `[int,1]` Orden D de la segunda derivada.
- **deltaOrder** `[int,2]` Orden D de la primera derivada.

Vea también

ShiftedDelta, VectorialDataFilter

DiscreteCosineTransform

Nombre

`edu.gtts.sautrela.sp.DiscreteCosineTransform` - Transformada de coseno discreta.

Descripción

Obtiene la transformada de cosenos discreta (Discrete Cosine Transform, DCT) [198] de cada vector de entrada. Más concretamente, calcula la variante DCT-II:

$$c_i = \sqrt{\frac{2-\delta_{i0}}{N}} \cdot \sum_{j=0}^{N-1} x_j \cdot \cos\left(\frac{\pi i}{N} \cdot (j+0{,}5)\right) \quad i = 0, 1, \ldots, L-1$$

donde N es la dimensión del vector de entrada y L es el número de coeficientes de salida. Es posible además aplicar una transformación o *liftering*[9] a los coeficientes de salida, realzando los centrales y atenuando los periféricos:

$$\acute{c}_i = \left(1 + \frac{L}{2} \cdot \sin\frac{\pi i}{L}\right) \cdot c_i$$

La DCT cuenta con una buena capacidad de compactación de la energía en el dominio transformado, de tal manera que la mayor parte de la información queda concentrada en unos pocos coeficientes. Para el caso de señales de voz, y cuando la DCT es aplicada al logaritmo de los coeficientes de un banco de filtros (véase el Procesador MelLogFilterBank), obtiene, en sus primeros coeficientes, una representación decorrelada del tracto vocal (más concretamente la respuesta impulso del tracto vocal). Dichos coeficientes, denominados *coeficientes cepstrales*, contienen información relativa al locutor, la articulación, etc. Además, y al tratarse de coeficientes decorrelados, es posible implementar su modelado mediante matrices de covarianza diagonal.

Características

Tipo Transformador.

Entrada Flujo vectorial de datos reales.

Salida Flujo vectorial de datos reales.

Retardo —

[9] Al tratarse de un filtrado típicamente aplicado a los cepstrales, el término *lifter* surge de la inversión del término *filter*, análogamente a *cepstrum*, que proviene de la inversión de *spectrum*.

Parámetros

- **energyIncluded** `[boolean,true]` Determina si el coeficiente c_0 (proporcional a la energía) es incluido o no. En caso contrario, el primer valor del vector de salida corresponderá al coeficiente c_1.
- **filterNum** `[int,13]` Número L de coeficientes de salida.
- **liftered** `[boolean,false]` Determina si se aplica o no la transformación de *liftering* a los coeficientes de salida.

Vea también

MelLogFilterBank, PowerSpectrum

Dithering

Nombre

`edu.gtts.sautrela.sp.Dithering` - Ruido de baja amplitud.

Descripción

Añade ruido de baja amplitud a las muestras de la entrada. El ruido añadido sigue una distribución o densidad de probabilidad uniforme comprendida en el rango $[-A, +A]$, donde A es la amplitud de dicho ruido. El *dithering* es una sencilla técnica que evita problemas numéricos que pudieran surgir ante señales de entrada constantes (o de componentes frecuenciales constantes).

Características

Tipo Transformador.

Entrada Flujo genérico de datos.

Salida Flujo genérico de datos.

Retardo —

Parámetros

- **amplitude** `[double,1.0]` Amplitud del ruido. Solamente son modificados los datos numéricos, a los que les será añadido un valor aleatorio: un valor entero comprendido en el rango $[-A, +A]$ a los datos enteros, o un valor real comprendido en el rango $[-A, +A)$ a los datos reales.

Vea también

Gain, Preemphasis, Windowing

Gain

Nombre

`edu.gtts.sautrela.sp.Gain` - Ganancia.

Descripción

Aplica una ganancia lineal a los datos de entrada

Características

Tipo Transformador.

Entrada Flujo escalar de datos enteros o reales.

Salida Flujo escalar de datos enteros o reales.

Retardo —

Parámetros

- **gain** `[double,1.0]` Valor de la ganancia. El tipo de dato de entrada no es modificado: en caso de tratarse de un dato entero, el valor real resultante de la ganancia es redondeado a un valor entero.

Vea también

LiveEnergySegmentator, VoiceActivityDetector, VUMeter

GMMTrainer

Nombre

`edu.gtts.sautrela.wfsa.models.GMMTrainer` - Entrenador de GMMs.

Descripción

Re-estima los parámetros de un modelo de mezcla de Gaussianas (GMM, por sus siglas en inglés) cargada desde un recurso de datos. Permite realizar bien una re-estimación de máxima verosimilitud (ML, por sus siglas en inglés) o máxima probabilidad a posteriori (MAP, por sus siglas en inglés). Ofrece además dos mecanismos complementarios para la aceleración de la estimación de parámetros:

1. Estimación concurrente. El procesador puede paralelizar la re-estimación de los parámetros del GMM, generando hilos de ejecución que procesarán en paralelo una fracción del conjunto total de datos. Una vez finalizado el procesamiento de cada hilo, los resultados son recombinados para obtener la estimación final de los parámetros del GMM.

2. Estimación rápida. Mediante la técnica conocida como *fast scoring*, los estadísticos que dan lugar a la re-estimación de parámetros son aproximados haciendo uso únicamente de las N componentes más probables para cada muestra de entrada. Este mecanismo resulta especialmente efectivo para GMMs compuestos de un gran número de componentes.

Características

Tipo Analizador.

Entrada Flujo vectorial de datos reales.

Salida Flujo vectorial de datos reales.

Retardo —

Parámetros

- **chunkSize** `[int,300]` Tamaño de los bloques de datos a generar para el entrenamiento paralelo. Los vectores de datos de entrada son agrupados en bloques de tamaño `chunkSize` y almacenados en una cola concurrente compartida por cada hilo de entrenamiento. Al agrupar los datos en bloques, se aligera la sobrecarga derivada del acceso concurrente a la cola de datos.

- **mapCount** `[double,0.0]` Valor de la probabilidad previa acumulada para la re-estimación MAP (*maximum a posteriori*) de parámetros. El entrenamiento MAP de los parámetros de un GMM hace uso del concepto de *probabilidad previa acumulada*, cuyo valor corresponde al número de muestras asociado a la estimación

actual de los parámetros (véase Sección 5.3). Un valor muy superior al número de muestras que serán posteriormente utilizadas en la fase de re-estimación conllevará la casi nula modificación de los parámetros del GMM, mientras que el valor 0 (valor por defecto) equivale a un entrenamiento por máxima verosimilitud.

- **outputFile** `[File,out.gmm]` Ruta del fichero en el que volcar el GMM entrenado.
- **modelURL** `[URL,file:OPENDIALOG]` Localizador del recurso de datos que contiene el GMM a entrenar, o "`file:OPENDIALOG`" para seleccionar el recurso mediante un diálogo (interfaz gráfica).
- **threadNumber** `[int,numberOfProcessors-1]` Número de hilos de ejecución a crear para la re-estimación del GMM. Cada hilo hace uso de un clon (copia) del GMM original para evitar así los problemas de consistencia de memoria y exclusión mutua derivados del acceso a recursos compartidos. Cada hilo consume bloques de datos de una cola concurrente (véase parámetro `chunkSize`) y realiza de manera aislada, y únicamente en base a los datos a los que accede, el proceso de estimación de los estadísticos que darían lugar a una re-estimación de su copia. Una vez consumidos todos los bloques de datos, finalizan los hilos de re-estimación, y los parámetros del GMM original son re-estimados en base al conjunto de estadísticos adquiridos por cada una de las copias. El valor por defecto del parámetro corresponde al número de núcleos lógicos menos uno (en sistemas con hyperthreading activado, el número lógico de núcleos es el doble del número real de ellos).
- **topN** `[int,0]` Número de componentes Gaussianas a utilizar para la estimación rápida (*fast scoring*) [147], o 0 para desactivar la estimación rápida. Dada una observación de entrada, la técnica denominada *fast scoring* selecciona de manera aproximada las N componentes con mayor responsabilidad[10], y utiliza únicamente esas N componentes para la estimación de los posteriores de cada componente (el posterior del conjunto de componentes excluidas se considera nulo). Esta técnica resulta especialmente interesante cuando se trabaja con GMMs de tamaño considerable, resultando posible acelerar el proceso de re-estimación sin perjuicio alguno. Por ejemplo, un GMM de 1024 componentes puede ser estimado de manera eficiente utilizando únicamente las 20 componentes más relevantes[11].

Vea también

Trainer

[10] Dado un GMM compuesto de K componentes con distribución de probabilidad $\mathcal{N}(\mu_i, \Sigma_i)$ y un vector de entrada $\mathbf{x}$, el ránking de las N componentes más relevantes puede ser aproximado en base al valor $(\mathbf{x}-\mu_i)\,\Sigma_i^{-1}\,(\mathbf{x}-\mu_i)^T$, que para el caso de covarianzas diagonales puede ser calculado eficientemente.

[11] Este debe entenderse como un ejemplo cualitativo que no tiene por qué cumplirse para todas las posibles situaciones. En función de la naturaleza de los datos y los parámetros actuales del GMM, podría variar la relación aquí mostrada. El método de *fast-scoring* supone que existirá un número N de componentes principales a partir del cual la responsabilidad acumulada del resto de componentes será despreciable.

Gaussianization

Nombre

`edu.gtts.sautrela.sp.Gaussianization` - Gaussianización.

Descripción

Aplica una transformación de gaussianización a cada una de las dimensiones de los parámetros de entrada, de tal manera que los parámetros de salida sigan una distribución normal (Gaussiana). La gaussianización (también conocida como *feature warping*) es una técnica de deformación de la función de densidad de probabilidad de una variable para adaptarla a otra función de densidad diferente. Sea CDF_{source} la función de distribución acumulada (CDF, de sus siglas en inglés) de la densidad original, y CDF_{target} la función de distribución acumulada de la densidad objetivo. Entonces, la función

$$f : x \rightarrow y = f(x) \ , CDF_{target}(y) = CDF_{source}(x)$$

transforma la densidad de probabilidad original en la densidad de probabilidad objetivo. La deformación de densidades de probabilidad tiene la propiedad de ser invariante a cualquier transformación monótona $t(x)$ del espacio de entrada, dado que dichas transformaciones no modifican los valores de la función de distribución acumulada:

$$CDF_{source}(x) = CDF_{t(source)}(t(x))$$

En el caso de la gaussianización, la función de densidad de probabilidad objetivo corresponde a una distribución normal con media 0 y varianza 1 (la Figura C.4 muestra un ejemplo de gaussianización). La transformación es estimada a partir de cada secuencia completa de datos de entrada, lo que añade un retardo de secuencia completa. En procesamiento de señales de voz, la gaussianización suele ser aplicada sobre los coeficientes cepstrales, de manera análoga a la normalización cepstral (véase el Procesador MeanVarianceNormalization), e incrementa la robustez frente al canal, el ruido aditivo y, en cierta medida, los efectos no lineales debidos a los transductores (micrófonos) [119, 224].

Características

Tipo Transformador.

Entrada Flujo vectorial de datos reales.

Salida Flujo vectorial de datos reales.

Retardo Secuencia completa de datos.

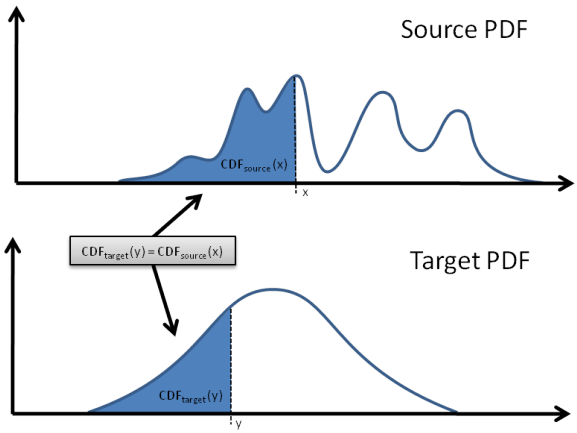


Figura C.4 Transformación de gaussianización. A cada valor original x le corresponde el valor y cuya probabilidad acumulada, $CDF_{target}(y)$, corresponda a la probabilidad acumulada de x en la distribución original, $CDF_{source}(x)$.

Parámetros

Vea también

LiveGaussianization, LiveMeanNormalization, MeanVarianceNormalization

GLDSKernel

Nombre

`edu.gtts.sautrela.sp.GLDSKernel` - Generalized Linear Discriminant Sequence Kernel.

Descripción

Estima o aplica una expansión polinomial normalizada que, dada una secuencia de vectores de entrada, genera un único vector de salida de dimensión fija (independientemente de la longitud de la secuencia de entrada).

Dado un vector de entrada $\mathbf{x} = [x_1, \ldots, x_n]$ de dimensión n, la expansión polinomial de orden d genera como resultado un vector $\mathbf{y} = [y_1, \ldots, y_m]$ de dimensión m donde los valores $y_i = x_{i_1} \cdot x_{i_2} \cdot \ldots \cdot x_{i_k}$ corresponden a todos los posibles monomios de orden $k \leq d$. Esta expansión polinomial equivale a generar todos los posibles monomios de orden d dado el vector transformado de entrada $\tilde{\mathbf{x}} = [1, x_1, \ldots, x_n]$ de dimensión $n+1$. Esta segunda interpretación permite obtener de manera más sencilla el valor de la dimensión m, que corresponde al número de d-combinaciones con repetición tomadas de un conjunto con $n+1$ elementos [194]:

$$m = \left(\left(\begin{array}{c} n+1 \\ d \end{array} \right) \right) = \left(\begin{array}{c} n+d \\ d \end{array} \right) = \left(\begin{array}{c} n+d \\ n \end{array} \right) = \frac{(n+d)!}{n! \cdot d!}$$

El procesador calcula el vector expandido promedio, el cual es normalizado por la raíz cuadrada de la diagonal de la matriz de correlación. El producto escalar de dichos vectores equivale a un kernel GLDS (Generalized Linear Discriminant Sequence Kernel) [45], por lo que los vectores resultantes (un único vector de dimensión fija por cada secuencia de datos de entrada) suelen ser utilizados como muestras de entrada de una máquina de vectores soporte (SVM, Support Vector Machine) y una función de núcleo (kernel) lineal. Los kernels GLDS han sido tradicionalmente utilizados en sistemas de verificación del locutor y verificación de la lengua.

El Procesador `GLDSKernel` cuenta con dos modos de funcionamiento:

- `TRAIN`: El procesador se comporta como un `Analizador`, estimando los parámetros de normalización a partir de los datos de entrada (la robustez de la estimación de los parámetros de normalización dependerá de la cantidad de datos suministrados). Una vez finalizado el flujo de datos, el resultado (las inversas de la raíces cuadradas de los coeficientes de correlación) es volcado a un fichero de datos (opción `gldskFile`).
- `EVAL`: El procesador se comporta como un `Transformador`. Carga inicialmente el fichero de parámetros de normalización, y genera un único vector de salida para cada secuencia de datos de entrada.

Características

Tipo Analizador (`TRAIN`) | Transformador (`EVAL`).

Entrada Flujo vectorial de datos reales.

Salida Flujo vectorial de datos reales.

Retardo —— (`TRAIN`) | Secuencia completa de datos (`EVAL`).

Parámetros

- **gldskFile** `[File,gldsk.data]` Ruta del fichero que contiene los parámetros de normalización.
- **mode** `[TRAIN|EVAL,EVAL]` Modo de funcionamiento.
- **order** `[int,1]` Orden d de la expansión polinómica. Nótese que la dimensión de salida será igual a $\frac{(n+d)!}{n!\cdot d!}$. Por ejemplo, dado un vector de entrada de 13 cepstrales y sus derivadas ($n = 39$), las dimensiones del vector de salida serán 820, 11480 y 123410 para expansiones polinómicas de orden 2, 3 y 4 respectivamente.

Vea también

HTKResourceReader

Nombre

`edu.gtts.sautrela.htk.HTKResourceReader` - Carga un recurso de audio.

Descripción

Carga un recurso de datos en formato HTK a partir de su localizador, generando una única secuencia de datos. Añade a la cabecera el siguiente conjunto de propiedades:

- `ResourceURL [URL]` : Localizador del recurso.
- `Samples [int]` : Número de muestras.
- `SamplePeriod [int]` : Periodo de muestreo (en múltiplos de 100ns).
- `SampleDim [int]` : Dimensionalidad de las muestras.
- `ParameterKind [enum]` : Tipo de parámetros HTK (`WAVEFORM`, `LPC`, `LPREFC`, `LPCEPSTRA`, `LPDELCEP`, `IREFC`, `MFCC`, `FBANK`, `MELSPEC`, `USER`, `DISCRETE`).
- `Qualifiers[int mask]` : Máscara de Cualificadores HTK (`E, N, D, A, C, Z, K, O, V, T`).

Características

Tipo Generador.

Entrada —

Salida Flujo vectorial de datos reales (la secuencia de vectores de datos que han sido codificados en base al formato HTK).

Retardo —

Parámetros

- **audioURL** `[URL,file:OPENDIALOG]` Localizador del recurso de audio, o `"file:OPENDIALOG"` para seleccionar el recurso mediante un diálogo (interfaz gráfica).

Vea también

ADBReader,AudioResourceReader

LiveEnergySegmentator

Nombre

`edu.gtts.sautrela.sp.LiveEnergySegmentator` - Eliminador de silencios en tiempo real.

Descripción

Utilizando un sencillo criterio de umbral de energía, filtra los segmentos de audio que no contienen voz. Dada una ventana de análisis, se determina que dicha ventana contiene voz si su energía promedio no dista más de un umbral τ del máximo posible E_{max}. Inicialmente, se utiliza una ventana deslizante para localizar la primera zona de voz, y una vez localizada, se analizan las ventanas adyacentes hasta dar con una ventana que no sobrepase el umbral establecido. La Figura C.5 muestra gráficamente un ejemplo de aplicación del Procesador `LiveEnergySegmentator.` A diferencia del Procesador VoiceActivityDetector, que genera una máscara de actividad de voz de forma independiente para cada muestra de entrada, el Procesador `LiveEnergySegmentator` se basa en el análisis de la energía de segmentos contiguos de señal, descartando los segmentos de poca energía.

Características

Tipo Transformador.

Entrada Flujo escalar de datos enteros.

Salida Flujo escalar de datos enteros.

Retardo Una ventana de análisis de tamaño `windowSize`.

Parámetros

- **maxValue** `[integer,`$2^{15}-1$`]` Valor máximo de la amplitud de la señal. Su valor por defecto presupone muestras enteras de 16 bits. Es utilizado para obtener el valor máximo de energía de una ventana de señal, E_{max}, por lo que el segmentador presupone que este valor máximo de la amplitud coincide con el rango dinámico de la señal. De no ser así, la segmentación será errónea.

- **threshold** `[double,-70]` Umbral de energía τ (en dB). Dada la energía promedio máxima posible de una ventana de análisis, E_{max} (véase `maxValue`), se determina que una ventana contiene voz si su energía promedio cumple la condición: $E > E_{max} - \tau$.

- **segmentsStreamed** `[boolean,true]` Genera una nueva secuencia de datos para cada segmento de voz. Para minimizar retardos (no esperar hasta el final de la secuencia de entrada), la cabecera de la secuencia original es añadida a cada nueva secuencia generada, mientras que el finalizador de secuencia original solo

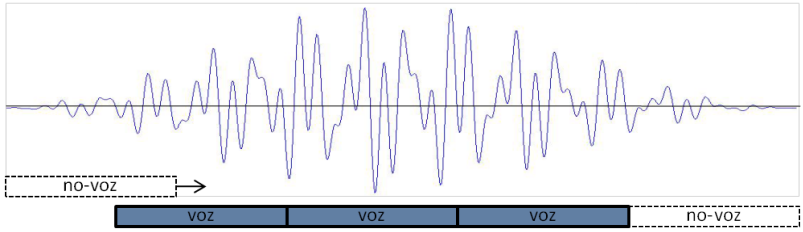


Figura C.5 Esquema de funcionamiento del procesador `LiveEnergySegmentator`. Una ventana deslizante localiza la primera zona de voz, a partir de la cual se analizan ventanas consecutivas.

es añadido a la última secuencia de voz. Si se desactiva esta opción, cada secuencia de datos de entrada da lugar a una única secuencia de salida que contiene la concatenación de segmentos de voz.[12]

- **windowSize** `[int,16000]` - Tamaño de la ventana de análisis. Su valor por defecto corresponde a 1 segundo en base a una frecuencia de muestreo de 16KHz.

Vea también

VoiceActivityDetector, Gain

[12] Si posteriormente va a realizarse algún procesamiento temporal, como derivadas o ShiftedDeltas, no es recomendable desactivar esta opción.

LiveGaussianization

Nombre

`edu.gtts.sautrela.sp.LiveGaussianization` - Gaussianización.

Descripción

Ofrece una alternativa en tiempo real al Procesador Gaussianization, utilizando un análisis de ventana deslizante para la estimación de la transformación (en vez de la secuencia completa de datos). Este procesamiento (comúnmente denominado *short-time Gaussianization*) es normalmente utilizado sobre los coeficientes cepstrales en tareas de verificación del locutor, con ventanas del orden de tres segundos. El uso de una ventana deslizante no solo reduce el retardo del procesador, sino que además le permite incrementar la robustez frente a variabilidades temporales en el canal, el ruido aditivo y, en cierta medida, los efectos no lineales debidos a los transductores (micrófonos) [119, 224].

Características

Tipo Transformador.

Entrada Flujo escalar de datos enteros.

Salida Flujo escalar de datos enteros.

Retardo Una ventana de análisis de tamaño `windowSize`.

Parámetros

- **windowSize** `[int,300]` - Tamaño de la ventana de análisis. Su valor por defecto corresponde a 3 segundos en base a una frecuencia de muestreo[13] de 100Hz.

Vea también

Gaussianization, LiveMeanNormalization, MeanVarianceNormalization

[13] Por frecuencia de muestreo nos referimos a la frecuencia de los parámetros de entrada del filtro, que en el caso de los coeficientes cepstrales suele ser típicamente 100Hz, tanto para grabaciones de 16Khz como señales telefónicas de 8Khz.

LiveMeanNormalization

Nombre

`edu.gtts.sautrela.sp.LiveMeanNormalization` - Normalización de media en tiempo-real.

Descripción

Ofrece una alternativa aproximada en tiempo real al Procesador MeanVarianceNormalization (únicamente en lo referente a la normalización de la media). Dado que la normalización de la media puede interpretarse como un filtro pasa-alto cuya frecuencia de corte ω_c se encuentra arbitrariamente cerca de 0, cabe plantearse el uso de otros filtros pasa-alto que muestren retardos mínimos. Tal es el caso del filtro exponencial con función de transferencia:

$$H(z) = (1-\alpha)\, z \cdot \frac{1 - z^{-1}}{1 - (1-\alpha) \cdot z^{-1}}$$

que equivale a una normalización donde la media $\overline{x}$ de los parámetros de entrada x es estimada en cada instante t mediante la ecuación:

$$\overline{x}_t = (1-\alpha)\, \overline{x}_{t-1} + \alpha x_t$$

El parámetro α permite definir la constante de tiempo τ del filtro, ya que dada una frecuencia de muestreo[14] f, ambas magnitudes se relacionan mediante la expresión:

$$\alpha = \frac{\ln 2}{\tau \cdot f}$$

Algunos autores indican que la constante de tiempo debe ser de al menos 5 segundos [79], si bien otros autores indican valores óptimos inferiores a los 700ms [17]. La Figura C.6 muestra la caracterización espectral del Procesador `LiveMeanNormalization` para diferentes valores del producto $\tau \cdot f$.

Características

Tipo Transformador.

Entrada Flujo vectorial de datos reales.

Salida Flujo vectorial de datos reales.

Retardo —

[14] Por frecuencia de muestreo nos referimos a la frecuencia de los parámetros de entrada del filtro, que en el caso de los coeficientes cepstrales suele ser típicamente 100Hz, tanto para grabaciones de 16Khz como señales telefónicas de 8Khz.

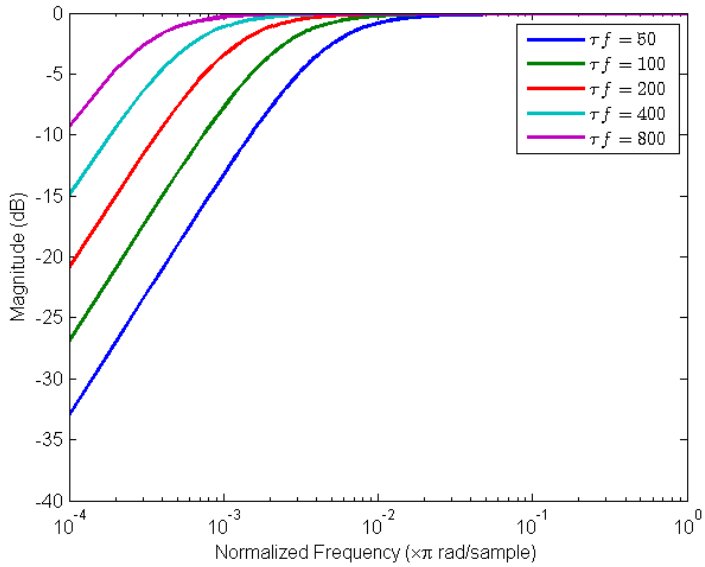


Figura C.6 Respuesta espectral del Procesador `LiveMeanNormalization` para diferentes valores del producto τf.

Parámetros

- **timeFreqConstant** `[int,500]` Valor del producto $\tau \cdot f$ (número de muestras correspondientes a la constante de tiempo τ del filtro). Su valor por defecto corresponde a la constante de tiempo $\tau = 5s$ en base a una frecuencia de muestreo[14] de 100Hz.

Vea también

MeanVarianceNormalization, RASTA

MeanVarianceNormalization

Nombre

`edu.gtts.sautrela.sp.MeanVarianceNormalization` - Normalización de media y varianza.

Descripción

Elimina la componente continua de los vectores de entrada (de cada una de sus dimensiones) y opcionalmente normaliza su varianza a 1. Los parámetros de normalización (la media y la varianza de los vectores) son estimados sobre la señal completa, por lo que el procesador añade un retardo de secuencia completa. Aplicado a los coeficientes cepstrales, equivale a la denominada Cepstral Mean Normalization (CMN) [21], la cual elimina de forma efectiva todo efecto debido a un filtro estacionario, como pueden serlo el tipo de micrófono, la distancia al micrófono o la acústica de la sala de grabación.

Características

Tipo Transformador.

Entrada Flujo vectorial de datos reales.

Salida Flujo vectorial de datos reales.

Retardo Secuencia completa de datos.

Parámetros

- **varianceToOne** `[boolean,true]` Determina si, además de restar la media, cada parámetro es dividido por su desviación típica, normalizando su varianza a 1.

Vea también

Gaussianization, LiveGaussianization, LiveMeanNormalization

MelLogFilterBank

Nombre

`edu.gtts.sautrela.sp.MelLogFilterBank` - Banco de filtros logarítmico con escala Mel.

Descripción

Obtiene el logaritmo de la salida de un banco de filtros triangulares basado en la escala Mel [189] como el que muestra la Figura C.7. La escala Mel, que suele aproximarse por la expresión

$$f_{mel} = 2595 * log(1 + f_{lin}/700)$$

es una escala no lineal que trata de imitar el comportamiento auditivo humano.

A pesar de que los valores por defecto del procesador hacen uso del rango máximo de frecuencias, en muchos casos resulta interesante reducir dicho rango. Así, en señales de 16000 Hz ($f_{nyquist} = 8000Hz$), es posible que exista muy poca información de interés más allá de los 6800Hz, e incluso en entornos con canales ruidosos pudiera ser interesante una frecuencia máxima de 5000Hz. En habla telefónica, por otro lado ($f_{nyquist} = 4000Hz$), el propio canal telefónico suele establecer frecuencias de corte en torno a 300Hz y 3700Hz. La documentación de CMU Sphinx4 (http://cmusphinx.sourceforge.net/sphinx4) muestra las siguientes configuraciones típicas:

Frecuencia de muestreo (Hz)	16000	11025	8000
nº filtros	40	36	31
minFreq	130	130	200
maxFreq	6800	5400	3500

Tabla C.1 Valores típicos para los parámetros de un banco de filtros Mel.

Características

Tipo Transformador.

Entrada Flujo vectorial de datos reales.

Salida Flujo vectorial de datos reales.

Retardo —

Parámetros

- **filterNum** `[int,40]` Número de filtros (tamaño del banco de filtros). Determina también la dimensión del vector de salida

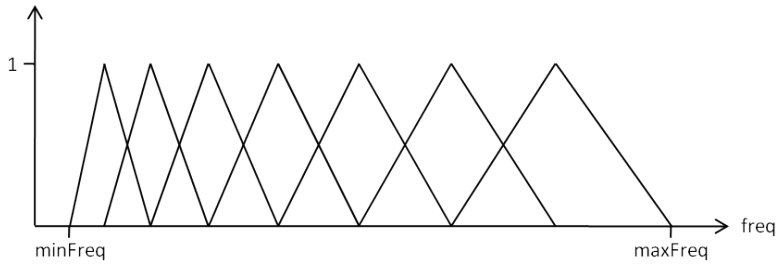


Figura C.7 Banco de filtros triangulares cuyas frecuencias centrales están dispuestas según la escala Mel. Los filtros son colocados en un rango de frecuencias $[minFreq, maxFreq]$, y las bases de cada filtro coinciden con las frecuencias centrales de los filtros adjacentes. Como puede deducirse de la imagen, las bajas frecuencias son enfatizadas frente a las altas.

- **maxFreq** `[double,6800.0]` Frecuencia máxima del banco de filtros. Debe encontrarse en el rango $(minFreq, F]$, donde

$$F = \frac{dim_{Entrada} - 1}{dim_{Entrada}} \cdot \frac{sampleRate}{2}$$

  es la frecuencia límite, siempre inferior a la frecuencia de Nyquist, $f_{nyquist} = sampleRate/2$. En caso de asignar un valor fuera de rango, se hace uso del valor máximo F.

- **minFreq** `[double,130.0]` Frecuencia mínima del banco de filtros.

- **sampleRate** `[int,16000]` Frecuencia de muestreo de los datos de entrada. Si la frecuencia de muestreo contenida en la cabecera del flujo de datos con coincide con la frecuencia suministrada, ocurrirá una excepción de procesamiento que finalizará la ejecución.

Vea también

PowerSpectrum, RASTA

PowerSpectrum

Nombre

`edu.gtts.sautrela.sp.PowerSpectrum` - Espectro de potencia.

Descripción

Obtiene el espectro de potencia para cada vector real de entrada mediante una transformada de Fourier (*Fast Fourier Transform*, FFT). El tamaño del vector de salida es igual a la mitad del tamaño de la FFT aplicada.

Características

Tipo Transformador.

Entrada Flujo vectorial de datos reales.

Salida Flujo vectorial de datos reales.

Retardo —

Parámetros

- **size** `[int,512]` El tamaño de la FFT (debe ser potencia de 2). Si la dimensión del vector de entrada es inferior, se rellena con ceros. Si por el contrario, es superior, únicamente se utilizarán los primeros *size* elementos. Los vectores de salida serán de dimensión $\frac{size}{2}$. Su valor por defecto permite calcular el espectro de potencia de ventanas de hasta 32ms en base a una frecuencia de muestreo de 16KHz.

Vea también

Windowing, MelLogFilterBank

Preemphasis

Nombre

`edu.gtts.sautrela.sp.Preemphasis` - Filtro de preénfasis.

Descripción

Aplica a la señal de entrada un filtro FIR pasa alto[15] con función de transferencia:

$$H(z) = 1 - \alpha \cdot z^{-1}$$

que trata de compensar la atenuación sufrida por las altas frecuencias durante la adquisición del audio. Suele ser utilizado en la primera fase de la parametrización, previo al ventaneo.

La Figura C.8 muestra la caracterización espectral del filtro de preénfasis para diferentes valores del factor α.

Características

Tipo Transformador.

Entrada Flujo escalar de datos enteros o reales.

Salida Flujo escalar de datos enteros o reales.

Retardo —

Parámetros

- **factor** `[double,0,97]` Factor α de preénfasis. El filtro se comporta como un filtro pasa alto únicamente para valores positivos del factor α.

Vea también

Dithering, Gain, Windowing

[15] El filtro de preénfasis se comporta como un filtro pasa alto únicamente para valores positivos del factor α.

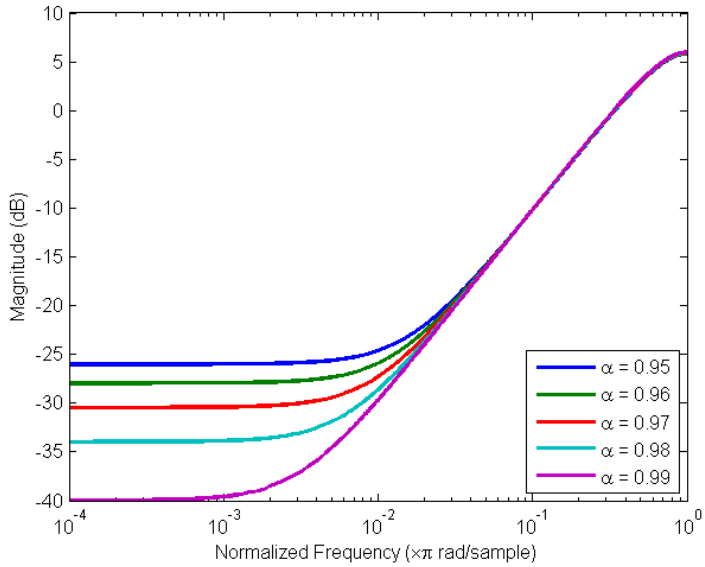


Figura C.8 Respuesta espectral de un filtro de preénfasis para diferentes valores del factor α.

ProgrammableProcessor

Nombre

`edu.gtts.sautrela.engine.util.ProgrammableProcessor` - Procesador programable.

Descripción

Este procesador ofrece la posibilidad de definir su comportamiento en tiempo de ejecución mediante la implementación del método `process` de la interfaz `DataProcessor` (véase Sección 3.2). Cuenta con un único parámetro, `code`, cuyo valor debe contener el código fuente de dicho método, expresado en lenguaje de programación Java. En el momento de instanciarse el procesador, el código suministrado es compilado, y en caso de no generar ningún error, es ejecutado. Deben tenerse en cuenta los siguientes aspectos:

- Una engine puede contener más de un `ProgrammableProcessor`. No existe ninguna limitación en lo que a su número se refiere.
- El código fuente suministrado es incorporado al cuerpo del método `process` de una clase que es compilada dinámicamente. El contenido exacto del código fuente de la clase resultante es el siguiente:

```
import edu.gtts.sautrela.engine.*;
import edu.gtts.sautrela.engine.data.*;
public class RAMDOM_UUID implements DataProcessor {
   public void process(Buffer in, Buffer out)
      throws DataProcessorException {
      CODE_GOES_HERE
   }
}
```

  Donde `RAMDOM_UUID` hace referencia a un identificador aleatorio. El código podrá utilizar las variables `in` y `out`, las cuales corresponden al Buffer de entrada y salida del Procesadsor, respectivamente.

- Solo son importadas las clases de los paquetes[16] `edu.gtts.sautrela.engine` y `edu.gtts.sautrela.engine.data`, por lo que en el resto de los casos deberá hacerse uso del nombre completo de clase.
- Dado que el parámetro `code` contiene el código fuente en lenguaje Java, y su sintaxis interfiere fácilmente con la del documento XML que lo ha de contener, es recomendable especificarlo dentro de una sección CDATA (véanse los ejemplos).
- El Procesador `ProgrammableProcessor` tiene por objeto permitir la creación de procesadores muy sencillos sin la necesidad de tener que generar un plugin[17]. No

[16] Las clases del paquete `java.lang` también son importadas en Java de manera automática.
[17] La Sección 3.2 incluye información adicional sobre cómo crear nuevos plugins.

obstante, la creación de `ProgrammableProcessor`-s complejos va claramente en contra de la filosofía de diseño de Sautrela y no es en absoluto recomendable. Debe tenerse en cuenta también que estos procesadores ni son parametrizables, ni cuentan con mecanismo de documentación alguno.

El Listado C.1 contiene un extracto de una engine que cuenta con un sencillo procesador programable, de tipo `Analizador`, que muestra por la salida estándar cada uno de los datos que recibe por el buffer de entrada (su funcionamiento es muy similar al del Procesador Sniffer). Nótese que el uso de una sección `CDATA` permite escribir el código fuente de Java sin preocuparse de posibles interferencias con la sintaxis XML. El Listado C.2 contiene también un extracto con otro procesador programable, en este caso de tipo `Generador`, que genera un flujo vectorial de datos reales aleatorios compuesto por cinco secuencias de vectores de dos dimensiones. Nótese que es necesario utilizar el nombre completo de clase `java.util.Random`, dado que su clase no ha sido importada.

Listado C.1 `ProgrammableProcessor.eng` - Extracto del descriptor XML de una engine con un procesador programable. Su funcionamiento es muy similar al del Procesador `Sniffer`, ya que se limita a mostrar por la salida estándar las secuencias de datos que recibe a su entrada.

```
...
<Processor name="MySniffer">
   <param name="code"><![CDATA[
      Data d = null;
      do {
         d = in.read();
         System.out.println(d);
         out.write(d);
      } while (d != Data.EOS);
   ]]></param>
</Processor>
...
```

Listado C.2 `ProgrammableProcessor2.eng` - Extracto del descriptor XML de una engine con un procesador programable que genera 5 secuencias compuestas por 200 vectores reales aleatorios de dimensión 2.

```
...
<Processor name="MyRandomVectorGenerator">
   <param name="code"> <![CDATA[
      java.util.Random r = new java.util.Random();
      for(int i = 0; i < 5; i++) {
         out.write(new StreamBegin());
         for(int j = 0; j < 200; j++) {
            double[] v = new double[]{ r.nextGaussian(), r.nextGaussian() }
            out.write(new DoubleData(v));
         }
         out.write(new StreamEnd());
      }
      out.write(Data.EOS);
   ]]> </param>
</Processor>
...
```

Características

Tipo Indeterminado.

Entrada Indeterminada.

Salida Indeterminada.

Retardo Indeterminado.

Parámetros

- **code** `[String,""]` Código fuente en lenguaje de programación Java que implemente el método `process` de la interfaz `DataProcessor` (véase Sección 3.2).

Vea también

Quantizer

Nombre

`edu.gtts.sautrela.vq.Quantizer` - Cuantificador vectorial

Descripción

Realiza una cuantificación vectorial de las muestras de entrada en base a un codebook (conjunto de centroides) que carga desde un recurso de datos. Dado un vector de entrada, el procesador vuelca un dato entero con el índice (con base 0) del centroide más próximo al vector de entrada.

Características

Tipo Transformador.

Entrada Flujo vectorial de datos reales.

Salida Flujo vectorial de datos reales.

Retardo —

Parámetros

- **binaryCdbk** `[boolean,false]` Determina si el formato del recurso de datos que contiene el codebook es binario (`true`) o XML (`false`).
- **cdbkURL** `[URL,file:OPENDIALOG]` Localizador del recurso de datos que contiene el codebook, o "`file:OPENDIALOG`" para seleccionar el recurso mediante un diálogo (interfaz gráfica). Su formato dependerá del parámetro `binaryCdbk`, pero en todo caso el recurso de datos debe contener un flujo vectorial de datos reales en formato Sautrela (véase el Procesador StreamWriter), donde cada vector representa un centroide del codebook

Vea también

CodebookTrainer

RandomNumberGenerator

Nombre

`edu.gtts.sautrela.engine.util.RandomNumberGenerator` - Generador aleatorio de números.

Descripción

Genera secuencias pseudoaleatorias de datos vectoriales (enteros o reales) cuyos valores siguen una distribución uniforme en un rango de valores. Su comportamiento pseudoaleatorio es acorde al mecanismo de semilla común de Sautrela (véase opción `-commonSeed` del Comando sautrela).

Características

Tipo Generador.

Entrada —

Salida Flujo vectorial de datos enteros/reales.

Retardo —

Parámetros

- **dataDim** `[int,1]` Dimensión de los vectores a generar.
- **dataType** `[INT|DOUBLE,DOUBLE]` Tipo de dato a generar.
- **maxValue** `[double,1.0]` Valor máximo del rango de valores a generar.
- **minValue** `[double,0.0]` Valor mínimo del rango de valores a generar.
- **streamLength** `[int,10]` Número de datos numéricos (vectores) a generar por secuencia.
- **streamNumber** `[int,5]` Número de secuencias a generar.

Vea también

RandomSymbolGenerator

RandomSymbolGenerator

Nombre

`edu.gtts.sautrela.wfsa.RandomSymbolGenerator` - Generador aleatorio de símbolos.

Descripción

Dado un autómata de estados finitos ponderado (WFSA, por sus siglas en inglés) cargado desde un recurso, obtiene un conjunto de secuencias aleatorias de símbolos. Independientemente de la naturaleza de los símbolos, que de penderá del modelo utilizado, cada símbolo es convertido a su representación textual, por lo que volcará a su salida una secuencia de datos textuales.

Su comportamiento pseudoaleatorio es acorde al mecanismo de semilla común de Sautrela (véase opción `-commonSeed` del Comando sautrela).

Características

Tipo Generador.

Entrada —

Salida Flujo de datos de texto.

Retardo —

Parámetros

- **streamNumber**`[int,1]` Número de secuencias a generar. La longitud de cada secuencia dependerá de la naturaleza propia del WFSA y será, presumiblemente, variable.
- **modelURL** `[URL,file:OPENDIALOG]` Localizador del recurso que contiene el WFSA a utilizar, o `"file:OPENDIALOG"` para seleccionar el recurso mediante un diálogo (interfaz gráfica).

Vea también

RandomNumberGenerator

RASTA

Nombre

`edu.gtts.sautrela.sp.RASTA` - Filtro RASTA.

Descripción

Aplica un filtrado RASTA (RelAtive SpecTrA) [78] a los parámetros de entrada (a cada una de las dimensiones del vector de entrada). Se trata de un filtro IIR (Infinite Impulse Response) con función de transferencia:

$$H(z) = 0{,}1z^4 \cdot \frac{2 + z^{-1} - z^{-3} - 2z^{-4}}{1 - 0{,}98z^{-1}}$$

que actúa como un filtro paso-banda que trata, por un lado, de eliminar variaciones lentas debidas al canal y, por el otro lado, suavizar variaciones rápidas posiblemente debidas a ruidos. La Figura C.9 muestra la caracterización espectral de un filtro RASTA.

El filtrado RASTA fue originalmente concebido para el procesamiento PLP (Perceptual Linear Prediction), pero posteriormente ha sido utilizado también en parametrizaciones basadas en coeficientes cepstrales, bien a la salida del banco de filtros o directamente sobre los propios coeficientes [241]. En algunos casos, se ha podido comprobar que la normalización cepstral en tiempo-real (véase Procesador LiveMeanNormalization) resulta más efectiva [79].

Características

Tipo Transformador.

Entrada Flujo vectorial de datos reales.

Salida Flujo vectorial de datos reales.

Retardo —

Vea también

LiveMeanNormalization, MeanVarianceNormalization

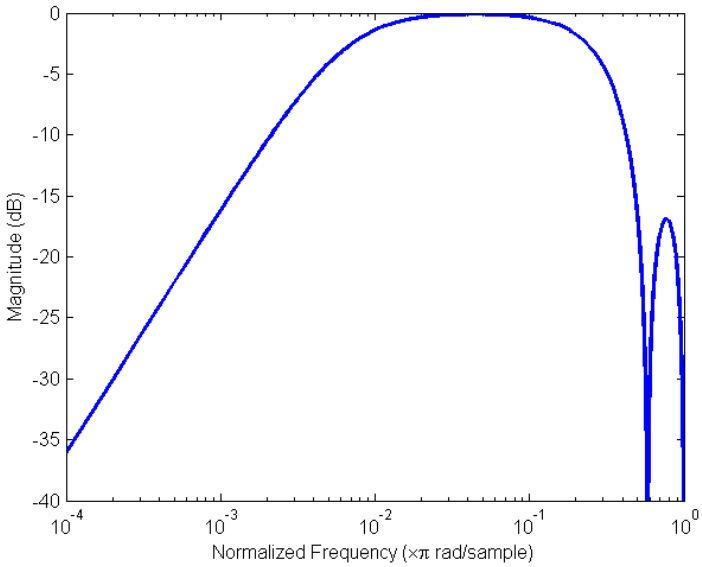


Figura C.9 Respuesta espectral del filtro RASTA.

RecognitionRate

Nombre

`edu.gtts.sautrela.wfsa.RecognitionRate` - Tasas de reconocimiento.

Descripción

Compara la salida de un decodificador frente a la transcripción de referencia contenida en las cabeceras de los datos y estima una tasa de reconocimiento que muestra por la salida estándar. Una función de evaluación o tasa de reconocimiento se expresa mediante una función de edición, la cual se define como el cociente de dos sumas de operaciones ponderadas de edición:

$$g(p) = \frac{R(p)}{Q(p)} = \frac{\alpha_s S + \alpha_i I + \alpha_b B + \alpha_a A}{\beta_s S + \beta_i I + \beta_b B + \beta_a A}$$

donde S, I, B y A se refieren al número de operaciones de edición (sustituciones, inserciones, borrados y aciertos, respectivamente) de un camino o alineamiento p entre la referencia y la hipótesis, y $\alpha = [\alpha_s, \alpha_i, \alpha_b, \alpha_a]$ y $\beta = [\beta_s, \beta_i, \beta_b, \beta_a]$ son los pesos que asignan el numerador y el denominador de la función de edición a cada una de las operaciones de edición. La obtención de una tasa equivale a la búsqueda del camino que optimice la función de edición asociada. Si la tasa representa un *grado de error*, entonces la optimización buscará el camino que minimice la función de edición, mientras que si la tasa representa un *grado de acierto*, la optimización buscará el camino que maximice la función de edición. No obstante, y dado que para toda función de edición $g(p)$ que representa un grado de acierto existe su versión grado de error equivalente, $f(p) = 1 - g(p)$, es posible referirse siempre al caso de funciones de edición que representan grados de error, comúnmente denominadas *distancias de edición*.

A menudo, el denominador $Q(p)$ de una distancia de edición no suele depender del camino p, sino que suele corresponder a la longitud de la referencia, $S + B + A$, a la longitud de la hipótesis, $S + I + A$, o puede tratarse de una combinación lineal de ambas, $(r + h) \cdot S + h \cdot I + r \cdot B + (r + h) \cdot A$. En tales casos, se dice que se trata de una distancia de edición no normalizada, ya que la optimización no depende del denominador (suele utilizarse también el término de distancia post-normalizada). En tales casos, la búsqueda del camino óptimo puede implementarse mediante la programación dinámica, que buscará el camino óptimo en base al numerador, $R(p)$. Sin embargo, si se trata de una distancia normalizada, el problema de optimización debe ser resuelto mediante la programación fraccional, que tendrá en cuenta el cociente $R(p)/Q(p)$ en el proceso de búsqueda del camino óptimo [109, 187, 144].

Características

Tipo Analizador.

Entrada Flujo de datos de texto.

Salida Flujo de datos de texto.

Retardo —

Parámetros

- **excludeSymbolPattern** `[String,""]` Los símbolos que coincidan con este patrón (expresión regular) son excluidos de la hipótesis y la referencia, con lo que no son tenidos en cuenta en la estimación de la tasa. Puede utilizarse para excluir de la estimación cierto conjunto de símbolos como los relativos a silencios y fenómenos de habla espontánea, o para restringir las tasas a un subconjunto de unidades como las vocales, etc.

- **matrixFile** `[File,null]` Ruta del fichero donde volcar la matriz de confusión (el valor por defecto no vuelca la matriz).

- **rate** `[rEP|rCP|ER|ACC|sdP|cP|wER|wACC,ACC]` Tasa a estimar. En la literatura pueden encontrarse infinidad de funciones de evaluación, a menudo denominadas de maneras diferentes [99, 111, 22, 84, 46, 116, 218]. A continuación se describen las diferentes tasas que soporta el presente procesador:

  - rEP (real Error Percentage): $\frac{(S+I+B)}{(S+I+B+A)} = 1 - rCP$. Porcentaje Real de Errores. Se trata de un porcentaje que no ignora ningún tipo de error. Una tasa del 0 % se obtiene en ausencia de errores, mientras que el 100 % se obtiene en ausencia de aciertos. Se basa en una distancia de edición normalizada.
  - rCP (real Correct Percentage): $\frac{A}{(S+I+B+A)} = 1 - rEP$. Porcentaje Real de Aciertos. Se trata de la versión grado de acierto del Porcentaje Real de Errores, REP.
  - ER (Error Rate): $\frac{(S+I+B)}{(S+B+A)} = 1 - ACC$. Índice de Error. Se trata de un índice no acotado superiormente que no ignora ningún tipo de error. Una tasa del 0 % se obtiene en ausencia de errores, mientras que, al depender del número de inserciones, su valor máximo no está acotado. Se basa en una distancia de edición no normalizada en la que el denominador corresponde a la longitud de la secuencia de referencia.
  - ACC (Accuracy): $\frac{(A-I)}{(S+B+A)} = 1 - ER$. Índice de Precisión. Se trata de la versión grado de acierto del Índice de Error, ER.
  - sdP (Substitutions and Deletions Percentage): $\frac{(S+B)}{(S+B+A)} = 1 - cP$. Porcentaje de Sustituciones y Borrados. Se trata de un porcentaje que ignora los errores de inserción. Una tasa del 0 % se obtiene en ausencia de sustituciones y borrados (independientemente del número de inserciones), mientras que el 100 % se obtiene en ausencia de aciertos. Se basa en una distancia de edición no normalizada en la que el denominador corresponde a la longitud de la secuencia de referencia.
  - cP (Correct Percentage): $\frac{A}{(S+B+A)} = 1 - sdP$. Porcentaje de Aciertos. Se trata de la versión grado de acierto del Índice de Error, ER.

- wER (weighted Error Rate): $\frac{(S+0{,}5\cdot I+0{,}5\cdot B)}{(S+B+A)} = 1 - wACC$. Índice de Error Ponderado. Similar al Índice de Error, ER, se trata de un índice no acotado superiormente que asigna el mismo coste a una sustitución y un par borrado-inserción. Una tasa del 0 % se obtiene en ausencia de errores, mientras que, al depender del número de inserciones, su valor máximo no está acotado. Se basa en una distancia de edición no normalizada en la que el denominador corresponde a la longitud de la secuencia de referencia. No debe confundirse con el término Word Error Rate, que no es otra cosa que el Índice de Error (Error Rate, ER) en el contexto de secuencias de palabras.
- wACC (weighted Accuracy): $\frac{(A-0{,}5\cdot I+0{,}5\cdot B)}{(S+B+A)} = 1 - wER$. Índice de Acierto Ponderado. Se trata de la versión grado de acierto del Índice de Error Ponderado, WER.

- **transcPropertyName** `[String,""]` Nombre de la propiedad a extraer de la cabecera de cada flujo de datos y que contiene la transcripción de referencia. Los símbolos contenidos en la transcripción de referencia deben ser coherentes con los símbolos recibidos a la entrada. Si, por ejemplo, la entrada corresponde a una secuencia de fonemas, la transcripción también deberá representar secuencias de fonemas.
- **transcPropertySplit** `[String,"\s"]` Patrón (expresión regular) utilizada como delimitador de símbolo para dividir (trocear) la transcripción de referencia y obtener la lista de símbolos de referencia.
- **vocURL** `[URL,null]` Localizador del vocabulario de símbolos. Si es nulo, el vocabulario es construido a partir de los símbolos contenidos en los conjuntos de hipótesis y referencias.

Vea también

Decoder

ShiftedDelta

Nombre

`edu.gtts.sautrela.sp.ShiftedDelta` - Derivadas concatenadas.

Descripción

Obtiene la derivada de los parámetros de entrada para el instante actual e instantes posteriores, agrupándolas en un solo vector. Aplicado a los coeficientes cepstrales, equivale al denominado Shifted Delta Cepstrum (SDC) [94][178]. Evalúa la misma expresión que el Procesador Delta pero, a diferencia de este, solo se calcula la primera derivada y cada vector de salida contiene un conjunto de derivadas calculadas a partir del instante actual. La Figura C.10 muestra el proceso de extracción de las derivadas.

Al añadir derivadas calculadas en torno al instante actual, los parámetros de salida contienen una mayor información dinámica. No obstante, debe tenerse en cuenta que la dimensionalidad de los vectores de salida crece de forma significativa: dado un vector de entrada de dimensión D, la dimensión del vector de salida será $D \cdot k$ o $D \cdot (k+1)$, en función de si los coeficientes estáticos son o no añadidos al vector de salida (parámetro `featurePrefixed`).

Características

Tipo Transformador.

Entrada Flujo vectorial de datos reales.

Salida Flujo vectorial de datos reales.

Retardo $(k-1) \cdot p + 2d \approx k \cdot p$ muestras.

Parámetros

- **d** `[int,1]` Orden de la derivada. Las derivadas son calculadas de forma análoga a la descrita en el Procesador Delta.
- **featurePrefixed** `[boolean,false]` Determina si se añade al vector de salida los n coeficientes estáticos (los valores del vector de entrada).
- **k** `[int,7]` Número de derivadas a calcular/concatenar.
- **n** `[int,7]` Dimensión efectiva de los datos de entrada. Puede ser inferior a la dimensión real de los datos de entrada, en cuyo caso solo se procesan los n primeros.
- **p** `[int,3]` Intervalo entre derivadas consecutivas

Vea también

Delta, VectorialDataFilter

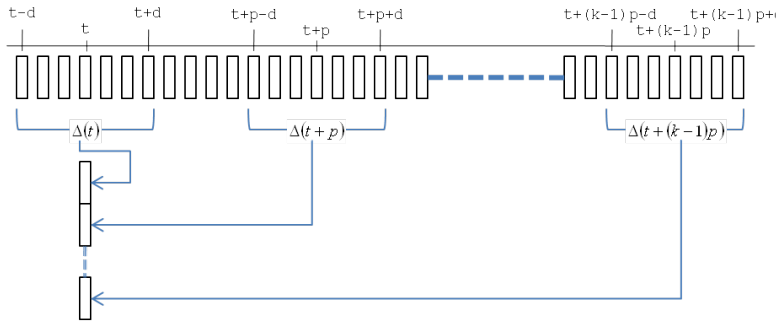


Figura C.10 Esquema de funcionamiento del Procesador `ShiftedDelta`. El vector de salida en un instante t contiene k derivadas de orden d calculadas a intervalos p.

Sniffer

Nombre

`edu.gtts.sautrela.engine.util.Sniffer` - Rastreador de datos.

Descripción

Analizador que muestra por la salida estándar las secuencias de datos. Estos pueden ser mostrados de dos modos distintos:

- Modo XML: Se muestran las secuencias completas de datos (marcadores, datos numéricos y datos de texto), haciendo uso de una representación XML para cada uno de los datos, y mostrando un dato por línea. El texto enviado a la salida estándar no conforma un documento XML, sino que cada dato de las secuencias es representado mediante un elemento XML (consúltese el Procesador StreamWriter en caso de desear generar dicho documento XML).
- Modo Texto: Solo se muestran los valores de los datos numéricos y de texto (un dato por línea), descartándose los marcadores `StreamBegin`, `StreamEnd` y `EOS`.

El siguiente texto podría ser el texto volcado por un Procesador Sniffer configurado para generar salida en formato XML, dado un flujo de datos compuesto por dos secuencias vectoriales de datos enteros:

```
<StreamBegin property1="value1_1" property2="value1_2"/>
<IntData value="0 1 2 3 4 5 6 7 8 9"/>
<IntData value="0 2 4 6 8 10 12 14 16 18"/>
<IntData value="1 2 4 8 16 32 64 128 256 512"/>
<StreamEnd/>
<StreamBegin property1="value2_1" property2="value2_2"/>
<IntData value="1 2 3 5 7 11 13 17 19 23"/>
<IntData value="0 1 1 2 3 5 8 13 21"/>
<IntData value="3 5 17 257 65537"/>
<StreamEnd/>
<EOS/>
```

El texto que se muestra a continuación podría corresponder al mismo caso, pero configurado para generar salida en formato de texto. Nótese que, al ignorar los marcadores, se mantiene el contenido de los datos, pero se pierde la estructura de las secuencias y la información que pudieran contener las cabeceras de las secuencias:

```
0 1 2 3 4 5 6 7 8 9
0 2 4 6 8 10 12 14 16 18
1 2 4 8 16 32 64 128 256 512
1 2 3 5 7 11 13 17 19 23
0 1 1 2 3 5 8 13 21
3 5 17 257 65537
```

Características

Tipo Analizador.

Entrada Flujo genérico de datos.

Salida Flujo genérico de datos.

Retardo —

Parámetros

- **mode** `[XML|TEXT,XML]` Determina si el modo de representación de los datos es XML (`XML`) o Texto (`TEXT`).

Vea también

DataPlotter, StreamGrep, StreamTester, StreamWriter

StreamGrep

Nombre

`edu.gtts.sautrela.engine.util.StreamGrep` - Selecciona secuencias de datos completas.

Descripción

Vuelca a su salida únicamente aquellas secuencias cuyas cabeceras contengan la propiedad indicada y su valor sea representado por el patrón (expresión regular) suministrado.

Características

Tipo Transformador.

Entrada Flujo genérico de datos.

Salida Flujo genérico de datos.

Retardo —

Parámetros

- **propertyName** `[String,"ResourceName"]` Nombre de la propiedad a usar para el filtrado. Todas aquellas secuencias cuya cabecera no contenga dicha propiedad no serán volcadas a la salida.
- **propertyValueRegex** `[String,".*"]` - Expresión regular que describe el patrón de selección. Las secuencias seleccionadas (volcadas a la salida) serán únicamente aquellas cuya cabecera contenga una propiedad denominada `propertyName` y cuyo valor coincida con el patrón `propertyValueRegex`.

Vea también

Sniffer, StreamHead, StreamSlicer, StreamTrimmer

StreamHead

Nombre

`edu.gtts.sautrela.engine.util.StreamHead` - Vuelca las secuencias de datos iniciales.

Descripción

Vuelca a su salida las primeras secuencias de datos o los primeros datos de cada secuencia (o ambas cosas a la vez).

Características

Tipo Transformador.

Entrada Flujo genérico de datos.

Salida Flujo genérico de datos.

Retardo —

Parámetros

- **numberOfData** `[int,10]` Número máximo de datos por secuencia a volcar a la salida, o -1 para volcar todos los datos. Solo contabilizan los datos numéricos y de texto (el marcador de inicio de secuencia, `StreamBegin`, no es tenido en cuenta).
- **numberOfStreams** `[int,10]` Número máximo de secuencias a volcar a la salida, o -1 para volcar todas las secuencias.

Vea también

StreamGrep, StreamSlicer, StreamTrimmer

StreamMixer

Nombre

`edu.gtts.sautrela.engine.util.StreamMixer` - Mezclador de secuencias.

Descripción

Toma el flujo de datos de entrada y lo mezcla o fusiona con un flujo binario de datos cargado desde un recurso (generado mediante un Procesador StreamWriter). Es preciso que se cumplan las siguientes condiciones:

- Ambos flujos deben estar compuestos por el mismo número de secuencias.
- Las secuencias a mezclar deben tener la misma longitud.
- Los datos a mezclar deben ser del mismo tipo.

La mezcla o fusión de datos se rige por las siguientes normas:

- La fusión de dos datos numéricos genera un nuevo dato numérico cuyo vector contiene los valores del dato de entrada y los valores del dato cargado, consecutivamente.
- La fusión de dos datos de texto da lugar a la concatenación del texto de entrada y el texto cargado, consecutivamente. La concatenación de ambas cadenas de caracteres hace uso del delimitador especificado.
- La fusión de marcadores de inicio y final da como resultado el marcador proveniente de la entrada (los marcadores provenientes del recurso cargado son descartados)

Este procesador puede ser utilizado para unir flujos de datos provenientes de una bifurcación en la línea de procesamiento de datos mediante el esquema de ramificación y confluencia de datos (véase Sección 4.1.2).

Características

Tipo Transformador/Generador.

Entrada Flujo genérico de datos.

Salida Flujo genérico de datos.

Retardo —

Parámetros

- **streamURL** `[URL,file:OPENDIALOG]` Localizador del recurso binario de datos, o "`file:OPENDIALOG`" para seleccionar el recurso mediante un diálogo (interfaz gráfica).
- **stringJoiner** `[String," "]` Cadena utilizada como delimitador en la concatenación de cadenas de caracteres.

Vea también

StreamReader, StreamWriter, VectorialDataFilter

StreamReader

Nombre

`edu.gtts.sautrela.engine.util.StreamReader` - Carga de flujo de datos.

Descripción

Carga un flujo de datos a partir de un localizador que hace referencia a un recurso compatible con el Procesador StreamWriter. Es posible la carga de recursos que hayan sido volcados en cualquiera de los dos modos, binario o XML. La carga de datos puede ser utilizada para:

- Importar secuencias de datos generadas a partir de software de terceros, dado que el formato XML resulta adecuado para el intercambio de información entre distintos sistemas o plataformas[18].
- Cargar el resultado de una fase inicial (y constante) de una engine que es ejecutada repetidamente. Tal es el caso de la fase de parametrización de una base de datos acústica en un contexto de entrenamiento de modelos acústicos. La parametrización puede repetirse en cada iteración de entrenamiento de los modelos, o, preferiblemente, realizarse una única vez, volcar la secuencia de datos resultante, y ser cargada posteriormente como entrada de la fase de entrenamiento de los modelos (procedimiento que se repetirá tantas veces como sea preciso).
- Crear una bifurcación en la línea de procesamiento de datos mediante el esquema de ramificación y confluencia de datos (véase Sección 4.1.2).

Características

Tipo Generador.

Entrada —

Salida Flujo genérico de datos.

Retardo —

Parámetros

- **binaryStream** `[boolean,true]` Determina si el recurso contiene datos en modo binario (`true`) o modo XML (`false`).
- **streamURL** `[URL,file:OPENDIALOG]` Localizador del recurso de datos, o "`file:OPENDIALOG`" para seleccionar el recurso mediante un diálogo (interfaz gráfica).

[18] Sautrela no incorpora ninguna herramienta de traducción de formatos XML que permita una importación desde ningún otro formato de software de terceros.

Vea también

StreamWriter, TextReader

StreamSlicer

Nombre

`edu.gtts.sautrela.engine.util.StreamSlicer` - Troceador de secuencias.

Descripción

Trocea cada secuencia de datos de entrada en secuencias de longitud fija, generando tantas nuevas secuencias como sean necesarias. Permite además descartar el resto final de cada secuencia.

Características

Tipo Transformador.

Entrada Flujo genérico de datos.

Salida Flujo genérico de datos.

Retardo En caso de descartar el resto final de cada secuencia de entrada, origina un retardo equivalente al tamaño de corte. Si no se descartan los restos finales, no añade retardo alguno.

Parámetros

- **discardTail** `[boolean,false]` En caso de activarse (`true`), se descarta el resto final de cada secuencia de entrada. De no ser activado (`false`), el resto final de cada secuencia de entrada genera una secuencia de salida de longitud inferior a la longitud de corte establecida.
- **streamLength** `[int,300]` Longitud de corte. Establece la longitud las secuencias de salida (a excepción del resto final, si `discardTail=false`).

Vea también

StreamGrep, StreamHead, StreamTrimmer

StreamTester

Nombre

`edu.gtts.sautrela.engine.util.StreamTester` - Analizador de la integridad de las secuencias de datos.

Descripción

Chequea la integridad de cada secuencia de datos, generando un error si:

1. La secuencia de datos no es conforme a:

   ```
   [StreamBegin (IntData|DoubleData|StringData)* StreamEnd]* EOS
   ```

2. Un dato transmitido es nulo (`null`).
3. Un dato numérico (`IntData` o `DoubleData`) contiene un vector nulo, o alguno de los valores del vector es erróneo ($\pm Inf$ o NaN).
4. Un dato de texto (`StringData`) contiene una cadena de caracteres nula.

El chequeador permite detectar dos tipos principales de errores:

- Errores de formato de secuencia (error de tipo 1). Son normalmente atribuibles a errores internos de algún procesador, que han de ser corregidos por el desarrollador.
- Errores de contenido (errores de tipo 2, 3 y 4). Pueden ser atribuibles tanto a errores propios de los procesadores, como a errores en la configuración de los procesadores o a errores en los datos de entrada.

Características

Tipo Analizador.

Entrada Flujo genérico de datos.

Salida Flujo genérico de datos.

Retardo —

Parámetros

No contiene parámetro alguno.

Vea también

DataPlotter, Sniffer

StreamTrimmer

Nombre

`edu.gtts.sautrela.engine.util.StreamTrimmer` - Eliminador de bordes.

Descripción

Descarta muestras al inicio o al final de cada secuencia de datos. Si el tamaño de la secuencia de entrada no es mayor que la suma de las muestras a descartar, se volcará una secuencia vacía.

Características

Tipo Transformador.

Entrada Flujo genérico de datos.

Salida Flujo genérico de datos.

Retardo Secuencia de datos completa.

Parámetros

- **discardAtBegin** `[int,0]` Número de muestras a descartar al inicio de cada secuencia de datos.
- **discardAtEnd** `[int,0]` Número de muestras a descartar al final de cada secuencia de datos.

Vea también

StreamGrep, StreamHead, StreamSlicer, VectorialDataFilter

StreamWriter

Nombre

`edu.gtts.sautrela.engine.util.StreamWrites` - Volcado de flujo de datos.

Descripción

Vuelca el flujo de datos a un fichero, el cual puede ser cargado posteriormente mediante el Procesador StreamReader. El volcado puede realizarse de acuerdo a los siguientes dos modos:

- Modo binario: Los datos son volcados en modo binario.
- Modo XML: Las secuencias de datos son volcadas a un documento XML. Este modo de volcado consume muchos más recursos, tanto computacionales como de almacenamiento, por lo que debería ser utilizado únicamente cuando sea preciso.

El volcado de datos puede ser utilizado para:

- Exportar las secuencias de datos resultantes de una engine, dado que el formato XML resulta adecuado para el intercambio de información entre distintos sistemas o plataformas[19].
- Ejecutar una única vez la fase inicial (y constante) de una engine que será ejecutada repetidamente. Tal es el caso de la fase de parametrización de una base de datos acústica en un contexto de entrenamiento de modelos acústicos. La parametrización puede repetirse en cada iteración de entrenamiento de los modelos, o, preferiblemente, realizarse una única vez, volcar la secuencia de datos resultante, y ser utilizada posteriormente como entrada de la fase de entrenamiento de los modelos (procedimiento que se repetirá tantas veces como sea preciso).
- Crear una bifurcación en la línea de procesamiento de datos mediante el esquema de ramificación y confluencia de datos (véase Sección 4.1.2).

Características

Tipo Analizador.

Entrada Flujo genérico de datos.

Salida Flujo genérico de datos.

Retardo —

[19] Sautrela no incorpora ninguna herramienta de traducción de formatos XML que permita una exportación a ningún otro formato de software de terceros.

Parámetros

- **binaryStream** `[boolean,true]` Determina si el volcado de datos es realizado en modo binario (`true`) o modo XML (`false`).
- **streamFile** `[File,"out.dump"]` Ruta del fichero de volcado.

Vea también

StreamReader

TextReader

Nombre

`edu.gtts.sautrela.text.TextReader` - Lector de texto.

Descripción

Carga un recurso de texto y genera una secuencia de datos de texto por línea. Ofrece la posibilidad de convertir el texto original a mayúsculas o minúsculas, filtrarlo y trocearlo. Los pasos que sigue el procesador son los siguientes:

1. Localiza el recurso de texto (parámetro `textURL`) y haciendo uso de la codificación indicada (parámetro `charset`), se extrae una cadena de caracteres por cada línea de texto. Cada cadena es procesada independientemente y da lugar a su propia secuencia de datos.
2. Opcionalmente, convierte la cadena de caracteres a mayúsculas (parámetro `toUpperCase`).
3. Opcionalmente, convierte la cadena de caracteres a minúsculas (parámetro `toLowerCase`).
4. Se eliminan de la cadena de caracteres todas las apariciones del patrón indicado en el parámetro `deleteRegex`.
5. La cadena de caracteres es troceada haciendo uso del patrón indicado en el parámetro `splitRegex`. Cada uno de los segmentos resultantes conforma un dato (`StringData`) que es volcado a la salida.

Características

Tipo Generador.

Entrada —

Salida Flujo de datos de texto.

Retardo —

Parámetros

- **charset** `[String,"UTF-8"]` Codificación del recurso de texto a cargar.
- **deleteRegex** `[String,""]` Expresión regular que describe el patrón del texto a eliminar. Las subsecuencias de texto que coincidan con el patrón indicado son eliminadas del texto original.
- **splitRegex** `[String,"\s"]` Expresión regular que describe el patrón delimitador. El patrón es utilizado para trocear cada línea del texto original, dando lugar a una secuencia de datos de texto.

- **textURL** `[URL,file:OPENDIALOG]` Localizador del recurso de texto, o "`file:OPENDIALOG`" para seleccionar el recurso mediante un diálogo (interfaz gráfica).
- **toLowerCase** `[boolean,false]` En caso de activarse, el texto de entrada es convertido a mayúsculas.
- **toUpperCase** `[boolean,false]` En caso de activarse, el texto de entrada es convertido a minúsculas.

Vea también

StreamReader

Trainer

Nombre

`edu.gtts.sautrela.wfsa.Trainer` - Entrenador genérico de WFSAs.

Descripción

Realiza una iteración del algoritmo EM (Expectation–Maximization) de re-estimación de los parámetros de un WFSA. También permite re-estimar un conjunto de capas de conocimiento, donde cada capa queda representada por un conjunto de WFSAs. En tal caso, y para cada secuencia de entrada, se generará una capa de supervisión basada en una transcripción contenida en la propia secuencia de datos.

De manera análoga al Procesador Decoder, es posible procesar la secuencia de entrada haciendo uso de una búsqueda en haz (*beam*), descartando en cada instante caminos que resulten poco probables y acelerando así de manera efectiva el proceso de re-estimación. En caso de una configuración de haz excesivamente restrictiva que provoque que no se obtenga ningún posible camino para una secuencia de entrada, el procesamiento de la secuencia es reiniciado, aumentando el haz de búsqueda.

Cada uno de los pasos del algoritmo EM es llevado a cabo por uno de los dos actores de la re-estimación: el Procesador `Trainer` y el WFSA en cuestión. El procesador estima la esperanza de todas las posibles transiciones a las que da lugar cada secuencia de datos de entrada (paso E), y dicha información es transmitida al autómata, quien se ocupa de realizar las estimaciones de máxima verosimilitud de los parámetros internos (paso M). Este desacoplamiento entre los dos pasos de re-estimación hace posible definir un mecanismo unificado de entrenamiento capaz de re-estimar todo modelo que implemente la interfaz de entrenamiento de Sautrela. Las Secciones 5.2, 5.3, 5.4 y 5.6 contienen una descripción más extensa del proceso unificado de re-estimación y desarrollan en profundidad el formalismo de los LMMs (Layered Markov Models), gracias a los cuales es posible integrar diferentes niveles de conocimiento en un único autómata, para así configurar complejos sistemas de re-estimación supervisada de conjuntos de modelos.

Características

Tipo Analizador.

Entrada Flujo genérico de datos.

Salida Flujo genérico de datos.

Retardo —

Parámetros

- **beam** `[double,-1.0]` Ancho de haz para la búsqueda de los posibles caminos, o un valor negativo para desactivar la búsqueda en haz. En cada instante (para

cada muestra de entrada) se descartan todos aquellos caminos cuyos logaritmos de probabilidad disten del camino más probable una cantidad superior al valor del haz.

- **beamRetryFactor** `[double,1.5]` Si el procesamiento de una secuencia de datos no da lugar a ningún camino posible (ninguno de los estados supervivientes es final), el procesamiento es reiniciado haciendo uso de un haz aumentado: $newBeam = beamRetryFactor \cdot oldBeam$.

- **beamRetryMax** `[int,10]` Número máximo de veces que se reintentará el procesamiento de la entrada con un haz aumentado. El hecho de no encontrar ningún camino posible, después de múltiples reintentos, puede indicar la existencia de algún error bien en los datos, bien en los modelos.

- **gcMinInterval** `[long,1000]` Determina el tiempo mínimo (en milisegundos) entre dos activaciones consecutivas del proceso de recolección de basura (véase parámetro `gcPolicy`).

- **gcPolicy** `[NONE|GC|FULLGC,NONE]` Determina la política de activación de recolección de basura implementada. La recolección de basura (del inglés *garbage collection*) es un mecanismo de gestión de memoria implementado por la JVM. Este mecanismo funciona de manera autónoma y normalmente no resulta preciso ni recomendable interaccionar con él, ya que se activa automáticamente bien de manera periódica o bien cuando detecta que los recursos de memoria empiezan a ser escasos[20]. Sin embargo, en situaciones en las que el uso de memoria es elevado y predecible, puede resultar conveniente la activación del mecanismo de recolección en instantes concretos. Tal es el caso del entrenamiento de modelos, donde el procesamiento de cada secuencia de entrada puede requerir una gran cantidad de memoria que puede ser liberada una vez finalizado este. La activación manual de la recolección de basura al finalizar el procesamiento de cada secuencia de entrada, en comparación con su activación automática, puede dar lugar a una mejor gestión de la memoria por parte de la JVM[21]. Las políticas de activación implementadas son:

  - `NONE`. No se aplica política alguna. La recolección de basura es guiada de manera automática por la propia JVM.
  - `GC`. Al finalizar el procesamiento de cada secuencia de entrada (Stream), y si ha transcurrido el tiempo mínimo establecido (véase parámetro `gcMinInterval`), se realiza una llamada al método de recolección de basura.

[20] Java implementa distintos mecanismos de recolección de basura, y lo aquí expuesto es una simplificación que en determinados casos puede resultar insuficiente, por lo que emplazamos al lector a obtener una información más detallada en [6].

[21] El rendimiento de las presentes políticas de activación del proceso de recolección de basura dependerá de la configuración de la JVM y el equipo físico en el que se ejecuta, pudiendo llegar a ser contraproducente en algunos casos. Este mecanismo está desactivado por defecto y debiera ser usado en aquellos casos en los que se detecten problemas de gestión de memoria por parte de la JVM. Nótese también que estas políticas de activación son complementarias a la configuración del mecanismo de recolección de basura descritas en [6].

  - `FULLGC`. Al finalizar el procesamiento de cada secuencia de entrada (Stream), y si ha transcurrido el tiempo mínimo establecido (véase parámetro gcMinInterval), se ejecuta un ciclo de llamadas consecutivas al método de recolección de basura hasta constatar que no se obtiene liberación alguna de espacio.

- **mapCount** `[double,0.0]` Valor de la esperanza previa acumulada para la re-estimación MAP (*maximum a posteriori*) de parámetros. El entrenamiento MAP de los parámetros es generalizado a todo WFSA mediante el concepto de *esperanza previa acumulada*, cuyo valor corresponde al número de muestras asociado a la estimación actual de los parámetros (véase Sección 5.3). Un valor muy superior al número de muestras que serán posteriormente utilizadas en la fase de re-estimación conllevará la casi nula modificación de los parámetros del modelo, mientras que el valor 0 (valor por defecto) equivale a un entrenamiento por máxima verosimilitud.

- **method** `[BAUMWELCH|VITERBI|INITIALIZATION,BAUMWELCH]` Método de re-estimación de parámetros. El procesador hace uso del algoritmo de re-estimación EM, con las siguientes tres variantes:

  - `BAUMWELCH`. Se trata del algoritmo *soft EM* en el que se alternan los pasos de esperanza (E) y maximización M. El paso E implica la estimación de los posteriores de probabilidad para el conjunto de transiciones en cada instante, mientras que el paso M es responsable de obtener la re-estimación de parámetros que maximice la verosimilitud en base a la estimación del paso anterior.
  - `VITERBI`. Se trata del algoritmo *hard EM* que, a diferencia del método *soft EM*, basa el paso E únicamente en la transición más probable. Para ello, se hace uso del camino más probable (aplicando el algoritmo de *Viterbi*), que determinará la transición más probable en cada instante.
  - `INITIALIZATION`. Se trata de un algoritmo similar a `BAUMWELCH`, salvo que los posteriores de probabilidad obtenidos en cada instante son normalizados para que resulten equiprobables. Este método puede resultar útil para re-inicializar un modelo.

- **modelURLList** `[String,"file:OPENDIALOG"]` Lista separada por comas de localizadores de recursos que contienen un WFSA o WFSASet a re-estimar. En función del valor del parámetro `transcPropertyName`, podrá variar el número de recursos, su contenido y el mecanismo de entrenamiento:

  - `transcPropertyName` = `null` : La lista consta de un único recurso que debe contener un WFSA que será entrenado a partir del conjunto completo de secuencias de datos.
  - `transcPropertyName` $\neq$ `null` : la lista puede estar compuesta de múltiples recursos, y todos ellos deben contener un WFSASet. Cada WFSASet corresponde a una capa de conocimiento (véase Sección 5.4) y representa un

conjunto de distribuciones de probabilidad sobre secuencias de unidades (modelos) de la capa directamente inferior, donde el primer recurso de la lista representa la capa del extremo superior y el último recurso representa la capa del extremo inferior. Para cada una de las secuencias de datos de entrada se añade una capa de supervisión a partir de una transcripción que debe encontrarse en la cabecera de datos (véase el conjunto de parámetros `transc*`).

- **outputFileList** `[String,"out.file"]` Lista separada por comas con rutas de fichero en los que volcar los recursos re-estimados. El tamaño de la lista debe coincidir con el tamaño de la lista suministrada a través del parámetro `modelURLList`.

- **transcInsertionMask** `[InsertionMask,]` Permite la emisión opcional de un símbolo especial (parámetro `transcInsertionSymbol`) al inicio de la transcripción (`l`, *left*), al final de la transcripción (`r`, *right*) o entre los símbolos de la transcripción (`i`, *inner*). Por ejemplo, si el símbolo especial correspondiese a un silencio, una máscara `"lr"` representaría la posibilidad de emitir silencios solo en los extremos, mientras que una máscara `"lir"` representaría la posibilidad de emitir silencios en cualquier sitio.

- **transcInsertionSymbol** `[String,""]` Nombre del símbolo especial que podrá ser insertado en la transcripción en función de la máscara determinada por el parámetro `transcInsertionMask`.

- **transcPropertyName** `[String,null]` Añade una capa de supervisión para cada secuencia de datos de entrada a partir de una transcripción contenida en la cabecera. El valor recuperado de la cabecera deberá ser una cadena de caracteres que representa la transcripción de la secuencia de datos como una secuencia de nombres de modelos de la capa superior suministrada (véase `modelURL`List).

- **transcPropertySplit** `[String,"\s"]` Expresión regular utilizada para trocear la transcripción. Cada segmento resultante deberá corresponder al nombre de alguno de los modelos cargados.

- **verbose** `[int,0]` Nivel de verbosidad utilizado para mostrar por la salida estándar información sobre el desarrollo de las diferentes fases del entrenamiento. Los niveles de verbosidad definidos son:

  0 No muestra nada

  1 En caso de entrenar un conjunto de modelos, muestra la transcripción de cada secuencia de entrada.

  2 Muestra la cabecera de cada secuencia de entrada y el tiempo dedicado al procesamiento de cada secuencia.

  3 Para cada instante (dato de entrada), muestra el tamaño del trellis (número de estados activos).

  4 Para cada instante (dato de entrada), muestra cada uno de los estados activos.

Vea también

Decoder, GMMTrainer

UnvoicedFeatureRemover

Nombre

`edu.gtts.sautrela.sp.UnvoicedFeatureRemover` - Elimina muestras que no corresponden a voz.

Descripción

Elimina las muestras (vectores de entrada) clasificadas como *no-voz* según se especifica en una máscara booleana contenida en la cabecera de cada secuencia de datos de entrada. Carga la máscara de actividad de voz (probablemente añadida por el Procesador VoiceActivityDetector, o también por el Procesador VADMaskLoader) y solo vuelca a la salida los datos vectoriales reales clasificados como *voz*. Chequea que la longitud de cada secuencia de datos coincida con la longitud de la máscara de actividad de voz de su cabecera, generando un error en caso contrario.

Características

Tipo Transformador.

Entrada Flujo vectorial de datos reales.

Salida Flujo vectorial de datos reales.

Retardo —

Vea también

VADMaskLoader, VoiceActivityDetector

VADMaskLoader

Nombre

`edu.gtts.sautrela.sp.VADMaskLoader` - Carga máscaras de actividad de voz desde un recurso de datos.

Descripción

Carga máscaras de actividad de voz desde un recurso de secuencias de datos en formato Sautrela (véanse los Procesadores StreamReader y StreamWriter) y las añade al flujo de datos de entrada. El recurso de datos cargado debe contener secuencias de datos compuestas por un único dato vectorial entero con valores `0/1` que representan la máscara de voz (0 =*no-voz* y 1 =*voz*) y cada cabecera de secuencia debe contener un identificador de recurso (propiedad `ResourceName`). Una vez cargadas las máscaras de actividad de voz desde el recurso de datos, se comienzan a procesar las secuencias de datos de entrada, añadiendo a cada una de ellas su correspondiente máscara de voz de acuerdo a su identificador de recurso. El orden de las secuencias cargadas desde del recurso de datos y el de las secuencias de entrada pueden diferir. No obstante, el procesador generará un error si se da alguna de las siguientes circunstancias:

- La cabecera de la secuencia de datos de entrada no contiene un identificador de recurso.
- El identificador del recurso de entrada no coincide con ninguno de los cargados desde el recurso de datos (no existe máscara de actividad de voz para el recurso de entrada).
- La diferencia de longitud entre la secuencia de datos y la máscara de actividad de voz supera el margen máximo establecido (véase parámetro `maxMargin`).

La secuencia de datos resultante contará en su cabecera con una propiedad, `VADFlag`, cuyo valor será una máscara booleana de actividad de voz (véase Procesador VoiceActivityDetector) que posteriormente podrá ser utilizada por el Procesador UnvoicedFeatureRemover para eliminar los vectores clasificados como *no-voz*.

El recurso de datos que contiene las máscaras podría haber sido generado con Sautrela, en cuyo caso habrá sido volcado mediante un Procesador StreamWriter, bien en modo binario, bien en modo XML. Si, por el contrario, procede de software de terceros, se tratará de un fichero de datos con formato XML compatible con los Procesadores StreamReader y StreamWriter.

Características

Tipo Analizador.

Entrada Flujo genérico de datos.

Salida Flujo genérico de datos.

Retardo Secuencia completa de datos.

Parámetros

- **binary** `[boolean,true]` Determina si el recurso contiene la máscara de actividad de voz en modo binario (`true`) o modo XML (`false`).
- **maskURL** `[URL,file:OPENDIALOG]` Localizador del recurso de datos, o "`file:OPENDIALOG`" para seleccionar el recurso mediante un diálogo (interfaz gráfica).
- **maxMargin** `[int,0]` Margen absoluto máximo de tolerancia ante diferencias de longitud entre la secuencia de datos y la máscara cargada. Dado que la máscara de actividad de voz podría provenir de una parametrización diferente a la de los datos de entrada, podría darse el caso de que no coincidieran exactamente en longitud. Si ambas longitudes no coinciden pero su diferencia no supera el margen establecido, los valores de la máscara son centrados y copiados en la máscara resultante, agregando valores de *no-voz* en los bordes si la longitud de la secuencia es mayor y descartando los valores de la máscara original en caso contrario. El procesador generará un error si la diferencia de longitud entre la secuencia de datos y la máscara de voz supera el margen máximo establecido.

Vea también

StreamReader, StreamWriter, VoiceActivityDetector, UnvoicedFeatureRemover

VectorialDataFilter

Nombre

`edu.gtts.sautrela.engine.util.VectorialDataFilter` - Filtro de componentes de vectores de datos.

Descripción

Modifica los datos vectoriales (`IntData` y `DoubleData`) de la entrada, realizando una reubicación o filtrado interno de los valores de cada dimensión del vector de entrada de acuerdo a la lista de índices proporcionada. Más concretamente:

- Los índices deben estar contenidos en el rango $[0, D-1]$, donde D es la dimensión del vector de entrada.
- La lista de índices puede estar compuesta por valores sencillos $(0, 1, 2, 3, 6, 7, 8, 9, 13)$ o por rangos $(0-3, 6-9, 13)$, en cuyo caso se refiere al rango ascendente de índices entre los dos valores (ambos inclusive).
- El tamaño del vector de salida es igual al tamaño de la lista expandida (sin rangos) de índices.
- La lista de índices define el orden de los valores en el vector resultante y, por tanto, no tiene por qué estar ordenada.
- La lista de índices puede contener índices repetidos, en cuyo caso, el vector resultante contendrá valores repetidos.

Características

Tipo Transformador.

Entrada Flujo genérico de datos.

Salida Flujo genérico de datos.

Retardo —

Parámetros

- **fieldList** `[string,""]` Lista separada por comas con los índices o rangos de indices a transmitir. El orden de salida es el indicado por la lista, y un mismo campo puede ser volcado múltiples veces. Es decir, la lista "3,1-5" generará la transmisión de los campos 3,1,2,3,4,5 respectivamente.

Vea también

StreamMixer

VoiceActivityDetector

Nombre

`edu.gtts.sautrela.sp.VoiceActivityDetector` - Detector de actividad de voz.

Descripción

Detecta la actividad de voz sobre una señal ya ventaneada, basándose en un criterio de energía. Dado el máximo de energía por ventana (en escala logarítmica) de una secuencia de datos, E_{max} y un umbral τ, aquellas muestras (vector de datos correspondiente a una ventana temporal) cuya energía es superior a $E_{max} - \tau$ son clasificadas como voz. Una vez procesada por completo una secuencia de entrada, vuelca esa misma secuencia a la salida, añadiendo a su cabecera (mediante una propiedad denominada `VADFlag`) una máscara booleana que contiene la información relativa a la actividad de voz. Dicha máscara puede ser utilizada posteriormente por el Procesador UnvoicedFeatureRemover, el cual elimina los vectores clasificados como *no-voz*.

Todo procesador que pueda ser colocado entre las fases de generación y posterior utilización de una máscara de actividad de voz (por ejemplo los Procesadores VoiceActivityDetector y UnvoicedFeatureRemover) y que altere la secuencia temporal implícita[22] en cada flujo de datos (bien por eliminación/agregación de muestras o su combinación), es responsable de mantener la sincronización entre las muestras y la máscara de actividad de voz de la cabecera[23]. En el caso concreto de los módulos propios de Sautrela, los Procesadores Delta y ShiftedDelta son los únicos que mantienen dicha sincronización, por lo que el resto de procesadores que alteran la secuencia temporal (por ejemplo los Procesadores DataJoiner, StreamHead, StreamSlicer y StreamTrimmer) nunca deberían colocarse entre dos fases de generación y utilización de máscaras de actividad de voz.

Características

Tipo Analizador.

Entrada Flujo vectorial de datos reales.

Salida Flujo vectorial de datos reales.

Retardo Secuencia completa de datos.

[22] Por secuencia temporal implícita nos referimos a la secuencia de marcas de tiempo asociada a cada dato. El Procesador StreamHead, por ejemplo, puede descartar todos los datos de cada secuencia excepto los iniciales. El Procesador ShiftedDelta, por el contrario, realiza una transformación basada en una ventana de análisis, por lo que cada vector resultante es producto de una combinación de un conjunto de vectores de entrada, y la frecuencia temporal de los vectores resultantes pudiera no coincidir con la frecuencia de los vectores de entrada (dependerá del desplazamiento de la ventana de análisis). De no modificarse de manera análoga la máscara de actividad de voz, se perderá la sincronización entre dicha máscara y los datos.

[23] La clase `edu.gtts.sautrela.sp.VoiceActivityDetector` contiene un método de utilidad, `transformVADFlag` que permite modificar la longitud de una máscara binaria mediante una transformación basada en ventana deslizante.

Parámetros

- **threshold** `[double,30]` Umbral de energía τ (en dB).

Vea también

Gain, LiveEnergySegmentator, VADMaskLoader, UnvoicedFeatureRemover

VUMeter

Nombre

`edu.gtts.sautrela.sp.VUMeter` - Medidor de nivel se señal.

Descripción

Muestra el nivel de señal en base al análisis de una ventana deslizante.

Características

Tipo Analizador.

Entrada Flujo escalar de datos enteros.

Salida Flujo escalar de datos enteros.

Retardo ——

Parámetros

- **maxValue** `[integer,`$2^{15}-1$`]` Valor máximo de la amplitud de la señal. Su valor por defecto presupone un muestreo de 16 bits.
- **windowSize** `[int,160]` - Tamaño de la ventana de análisis. Su valor por defecto corresponde a 10ms, suponinedo una frecuencia de muestreo de 16KHz.

Vea también

Gain, Sniffer, StreamTester

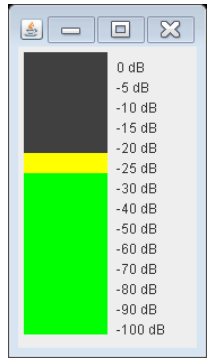


Figura C.11 El VUMeter muestra la energía de cada ventana de análisis frente a la energía máxima posible.

Windowing

Nombre

`edu.gtts.sautrela.sp.Windowing` - Ventaneo.

Descripción

Aplica un ventaneo [76, 215] al flujo escalar de datos de entrada, devolviendo a su salida un flujo vectorial de datos. Cada vector de salida contiene la secuencia de valores correspondientes a cada ventana de análisis, y su dimensión es igual al tamaño de ventana configurado. Es posible aplicar diversas funciones de ventana $v[i]$, $i = \{0 \dots N-1\}$:

- Rectangular o Dirichlet:
$$v[i] = 1$$
- Hamming:
$$v[i] = 0,54 - 0,46 \cdot \cos\left(\frac{2\pi i}{N-1}\right)$$
- Hann (Hanning):
$$v[i] = 0,5 \cdot \left(1 - \cos\left(\frac{2\pi i}{N-1}\right)\right)$$
- Blackman:
$$v[i] = 0,42 - 0,5 \cdot \cos\left(\frac{2\pi i}{N-1}\right) + 0,08 \cdot \cos\left(\frac{4\pi i}{N-1}\right)$$
- Blackman-Harris:
$$v[i] = 0,36 - 0,49 \cdot \cos\left(\frac{2\pi i}{N-1}\right) + 0,14 \cdot \cos\left(\frac{4\pi i}{N-1}\right) - 0,012 \cdot \cos\left(\frac{6\pi i}{N-1}\right)$$

Características

Tipo Transformador.

Entrada Flujo escalar de datos enteros o reales.

Salida Flujo vectorial de datos enteros o reales.

Retardo Una ventana de análisis de tamaño `size`.

Parámetros

- **cutDC** `[boolean,false]` Elimina la componente continua calculada sobre cada ventana.
- **function** `[NONE|SQUARE|HAMMING|HAN|BLACKMAN|BLACKMAN-HARRIS,HAMMING]` Función de ventaneo a aplicar. La función de ventaneo `NONE`, si los datos de entrada son enteros, genera a la salida un flujo vectorial de datos enteros (mantiene la naturaleza entera de los datos de entrada). En el resto de casos, se vuelca a la salida un flujo vectorial de datos reales.
- **shift** `[int,160]` Desplazamiento de la ventana deslizante. Su valor por defecto corresponde a 10ms, suponiendo una frecuencia de muestreo de 16KHz.
- **size** `[int,400]` Tamaño de la ventana deslizante. Su valor por defecto corresponde a 25ms, suponiendo una frecuencia de muestreo de 16KHz.

Vea también

ADBReader, AudioRecorder, AudioResourceReader, Gain, Dithering

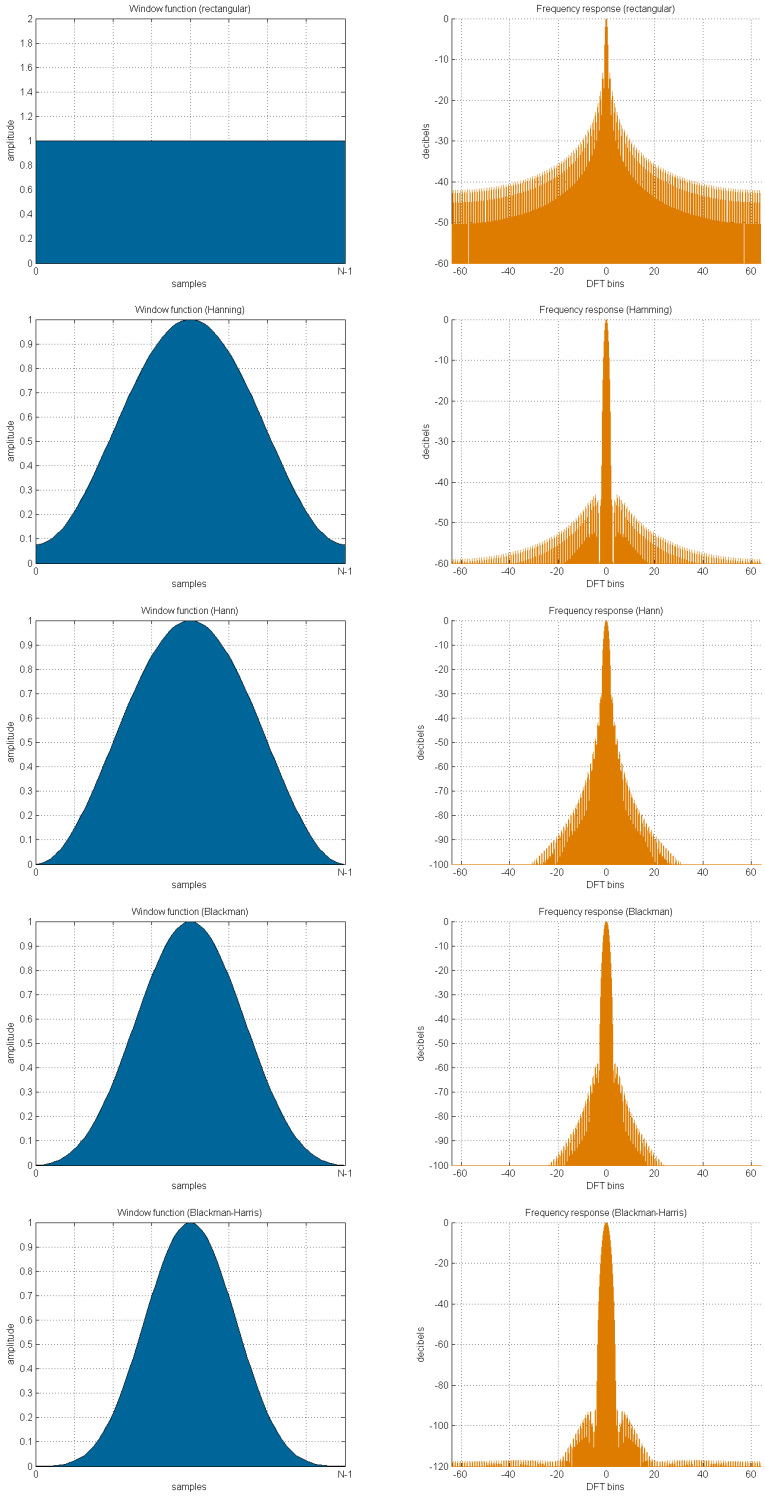


Figura C.12 Valor en amplitud y respuesta frecuencial de las ventanas rectangular, Hamming, Hann (Hanning), Blackman y Blackman-Harris.

Bibliografía

[1] EJML, Efficient Java Matrix Library. http://ejml.org. [Online; accessed 02-Dec-2015].

[2] Java Matrix Benchmark. https://github.com/lessthanoptimal/Java-Matrix-Benchmark. [Online; accessed 02-Dec-2015].

[3] la4j, Linear Algebra for Java. http://la4j.org. [Online; accessed 02-Dec-2015].

[4] LMAX Disruptor: High Performance Inter-Thread Messaging Library. http://lmax-exchange.github.io/disruptor. [Online; accessed 02-Dec-2015].

[5] OjAlgo, Open Source Java code to do mathematics, linear algebra and optimisation. http://ojalgo.org. [Online; accessed 02-Dec-2015].

[6] Oracle Technetwork: Java Garbage Collection Basics. http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html. [Online; accessed 02-Dec-2015].

[7] Parallel Colt, a multithreaded version of Colt. https://sites.google.com/site/piotrwendykier/software/parallelcolt. [Online; accessed 02-Dec-2015].

[8] The Checker Framework: Pluggable type checkers for Java. http://types.cs.washington.edu/checker-framework. [Online; accessed 02-Dec-2015].

[9] The Java Tutorials: Concurrency. http://docs.oracle.com/javase/tutorial/essential/concurrency. [Online; accessed 02-Dec-2015].

[10] The Java Tutorials: JavaBeans. http://docs.oracle.com/javase/tutorial/javabeans. [Online; accessed 02-Dec-2015].

[11] The Java Tutorials: Packaging Programs in JAR Files. http://docs.oracle.com/javase/tutorial/deployment/jar. [Online; accessed 02-Dec-2015].

[12] The Java Tutorials: Regular Expressions. http://docs.oracle.com/javase/tutorial/essential/regex. [Online; accessed 02-Dec-2015].

[13] The Java Tutorials: The Reflection API. http://docs.oracle.com/javase/tutorial/reflect. [Online; accessed 02-Dec-2015].

[14] UJMP, Universal Java Matrix Package. https://ujmp.org. [Online; accessed 02-Dec-2015].

[15] A. Abad, L. J. Rodriguez-Fuentes, M. Penagarikano, A. Varona, M. Diez, and G. Bordel. On the calibration and fusion of heterogeneous spoken term detection systems. In *Proceedings of Interspeech 2013*, Lyon, France, 25-29 aug. 2013.

[16] J. Alam, T. Kinnunen, P. Kenny, P. Ouellet, and D. O'Shaughnessy. Multi-taper MFCC and PLP features for speaker verification using i-vectors. *Speech Communication*, 55(2):237 – 251, 2013.

[17] B. Andrassy, F. Hilger, and C. Beaugeant. Investigations on the combination of four algorithms to increase the noise robustness of a DSR front-end for real world car data. In *IEEE Automatic Speech Recognition and Understanding Workshop*, Trento, Italy, Dec. 2001.

[18] X. Anguera, L. Rodriguez-Fuentes, A. Buzo, F. Metze, I. Szoke, and M. Penagarikano. Quesst2014: Evaluating query-by-example speech search in a zero-resource setting with real-life queries. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, pages 5833–5837, April 2015.

[19] X. Anguera, L. J. Rodriguez-Fuentes, F. Metze, Szoke, Buzo, and M. Penagarikano. Query-by-example spoken term detection on multilingual unconstrained speech. In *Proceedings of Interspeech 2014*, Singapore, 14-18 Sep 2014.

[20] X. Anguera, L. J. Rodriguez-Fuentes, I. Szoke, A. Buzo, F. Metze, and M. Penagarikano. Query-by-example spoken term detection evaluation on low-resource languages. In *Proceedings of the 4th International Workshop on Spoken Language Technologies for Under-resourced Languages (SLTU'14)*, pages 24–31, St. Petersburg, Russia, 14-16 May 2014.

[21] B. S. Atal. Effectiveness of linear prediction characteristics of the speech wave for automatic speaker identification and verification. *the Journal of the Acoustical Society of America*, (55), 1974.

[22] L. R. Bahl, R. Bakis, J. Bellegarda, P. F. Brown, D. Burshtein, S. K. Das, P. V. de Souza, P. S. Gopalakrishnan, F. Jelinek, D. Kanevsky, R. L. Mercer, A. J. Nadas, D. Nahamoo, and M. A. Picheny. Large vocabulary natural language continuous speech recognition. In *Acoustics, Speech, and Signal Processing (ICASSP), 1989 IEEE International Conference on*, pages 465–467, Glasgow, 1989.

[23] L. E. Baum. An inequality and associated maximization technique in statistical estimation for probabilistic functions of Markov processes. *Inequalities*, 3:1–8, 1972.

[24] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[25] A. W. Black and K. A. Lenzo. Flite: a small fast run-time synthesis engine. In *Proceedings of the 4th ISCA Workshop on Speech Synthesis*, Scotland, August-September 2001.

[26] G. Bordel. *Aprendizaje automatico de modelos K-Explorables estocásticos en Reconocimiento Automático del Habla.* PhD thesis, University oh The Basque Country, 1996.

[27] G. Bordel, A. Casillas, M. Penagarikano, L. J. Rodriguez-Fuentes, and A. Varona. A system architecture for multilingual spoken document retrieval. In *Actas de las V Jornadas en Tecnología del Habla*, pages 217–220, Bilbao, November 2008.

[28] G. Bordel, A. Casillas, M. Penagarikano, L. J. Rodriguez-Fuentes, and A. Varona. An XML resource definition for spoken document retrieval. In *Proceedings of the Iberian SLTech 2009*, 2009.

[29] G. Bordel, M. Diez, I. Landera, S. Nieto, M. Penagarikano, L. J. Rodriguez-Fuentes, A. Varona, and M. Zamalloa. Hearch: A multilingual spoken document retrieval system. In *IEEE Automatic Speech Recognition and Understanding Workshop (demo)*, 2009.

[30] G. Bordel, A. Ezeiza, K. de Ipina, J. López, M. Peñagarikano, and E. Zulueta. Language resources for a bilingual automatic index system of broadcast news in Basque and Spanish. In A. Sanfeliu and M. Cortés, editors, *Progress in Pattern Recognition, Image Analysis and Applications*, volume 3773 of *Lecture Notes in Computer Science*, pages 1047–1054. Springer Berlin Heidelberg, 2005.

[31] G. Bordel, A. Ezeiza, K. López de Ipiña, J. Lopez, M. Penagarikano, and E. Zulueta. Language resources for a bilingual automatic index system of broadcast news in Basque and Spanish. In *CIARP*, pages 1047–1054, 2005.

[32] G. Bordel, A. Ezeiza, K. López de Ipiña, D. Martinez, M. Mendez, M. Penagarikano, C. Tovar, and E. Zulueta. Digital resources for automatic speech recognition of broadcast news in Basque and Spanish. In *Proccedings of the International Symposium on Social Communication*, 2005.

[33] G. Bordel, N. Ezeiza, K. López de Ipiña, M. Mendez, M. Penagarikano, T. Rico, C. Tovar, and E. Zulueta. Development of resources for a bilingual automatic index system of broadcast news in Basque and Spanish. In *Proceedings of the fourth international conference on Language Resources and Evaluation, LREC*, 2004.

[34] G. Bordel, N. Ezeiza, K. López de Ipiña, M. Mendez, M. Penagarikano, T. Rico, C. Tovar, and E. Zulueta. Linguistic resources and tools for automatic speech recognition in Basque. In *Proceedings of the SALTMIL Workshop at LREC*, 2004.

[35] G. Bordel, S. Nieto, M. Penagarikano, L. J. Rodriguez-Fuentes, and A. Varona. Automatic subtitling of the Basque Parliament plenary sessions videos. In *Proceedings of Interspeech 2011*, Florence, Italy, 28-31 August 2011.

[36] G. Bordel, S. Nieto, M. Penagarikano, L. J. Rodriguez-Fuentes, and A. Varona. A simple and efficient method to align very long speech signals to acoustically

imperfect transcriptions. In *Proceedings of Interspeech 2012*, Portland, Oregon, USA, 9-13 sept. 2012.

[37] G. Bordel, M. Penagarikano, L. J. Rodriguez-Fuentes, A. Álvarez, and A. Varona. Probabilistic kernels for improved text-to-speech alignment in long audio tracks. *IEEE Signal Processing Letters*, 23(1):126–129, january 2016.

[38] G. Bordel, M. Penagarikano, L. J. Rodriguez-Fuentes, and A. Varona. Aligning very long speech signals to bilingual transcriptions of parliamentary sessions. In *Iberspeech 2012*, Madrid, Spain, 21-23 nov. 2012.

[39] G. Bordel, M. Penagarikano, L. J. Rodriguez-Fuentes, and A. Varona. OBAM-PV: una aplicación para el subtitulado de videos de Sesiones Plenarias del Parlamento Vasco. In *XXIX Congreso de la Sociedad Española para el Procesamiento de Lenguaje Natural (SEPLN)*, Madrid, 18-20 sep. 2013.

[40] G. Bordel, A. Varona, and M. Torres. K-tlss(s) language models for speech recognition. In *Acoustics, Speech and Signal Processing (ICASSP), 1997 IEEE International Conference on*, 1997.

[41] T. Bray, E. Maler, J. Paoli, F. Yergeau, and M. Sperberg-McQueen. Extensible markup language (XML) 1.0 (fifth edition). W3C recommendation, W3C, Nov. 2008. http://www.w3.org/TR/2008/REC-xml-20081126/.

[42] T. Bray, J. Paoli, M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (XML) 1.0 (fourth edition). W3C recommendation, W3C, Aug. 2006. http://www.w3.org/TR/2006/REC-xml-20060816.

[43] W. L. Buntine. Operations for learning with graphical models. *Journal of Artificial Intelligence Research*, 2:159–225, 1994.

[44] W. Campbell. Generalized linear discriminant sequence kernels for speaker recognition. In *Acoustics, Speech, and Signal Processing (ICASSP), 2002 IEEE International Conference on*, volume 1, pages I–161–I–164, May 2002.

[45] W. M. Campbell, J. P. Campbell, D. A. Reynolds, E. Singer, and P. A. Torres-carrasquillo. Support vector machines for speaker and language recognition. *Computer Speech and Language*, 20:210–229, 2006.

[46] M.-J. Castro, P. Aibar, F. Casacuberta, and E. Vidal. Automatic selection of sublexic templates by using dynamic time warping techniques. In L. Torres, E. Masgrau, and M. Ángel Lagunas, editors, *Signal Processing V: Theories and Applications*, pages 1351–1354. Elsevier Science Publishers B.V., 1990.

[47] S. Chen, S. Chen, S. Doh, M. Eskenazi, E. G. Ea, B. Raj, M. Ravishankar, R. Rosenfeld, M. Siegler, R. Stern, and E. Thayer. The 1997 CMU Sphinx-3 english broadcast news transcription system. In *In Proceedings of the 1998 DARPA Speech Recognition Workshop. DARPA*, pages 55–59, 1998.

[48] C. Chinrungrueng and C. Sequin. Optimal adaptive k-means algorithm with dynamic adjustment of learning rate. *IEEE Transactions on Neural Networks*, 6(1):157–169, Jan 1995.

[49] P. Clarkson and R. Rosenfeld. Statistical language modeling using the CMU-Cambridge toolkit. In G. Kokkinakis, N. Fakotakis, and E. Dermatas, editors, *Proceedings of Eurospeech*. ISCA, 1997.

[50] A. Conde, M. Larrañaga, J. Rubio, C. Vaquero, E. Irigoyen, K. L. de Ipiña, N. Garay, A. Ezeiza, A. Soraluze, M. Penagarikano, G. Bordel, L. J. Rodriguez-Fuentes, J. M. L. Guede, R. Martínez, M. Ezquerra, and O. Zubillaga. Laguntxo, a flexible intelligent tutoring system to improve the quality of life of the people with cognitive disabilities. Beurs van Berlage, Amsterdam, Holland, 2008.

[51] S. Davis and P. Mermelstein. Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 28(4):357–366, Aug 1980.

[52] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, 39(1):1–38, 1977.

[53] M. Diez, M. Penagarikano, G. Bordel, A. Varona, and L. J. Rodriguez-Fuentes. On the complementarity of short-time Fourier analysis windows of different lengths for improved language recognition. In *Proceedings of Interspeech 2014*, Singapore, 14-18 sep. 2014.

[54] M. Diez, M. Penagarikano, L. J. Rodriguez-Fuentes, A. Varona, and G. Bordel. University of the Basque Country system for the 2011 NIST SRE Analysis Workshop. In *NIST 2011 Speaker Recognition Analysis Workshop*, Atlanta (USA), 8-9 december 2011.

[55] M. Diez, M. Penagarikano, A. Varona, L. J. Rodriguez-Fuentes, and G. Bordel. GTTS system for the Albayzin 2010 Speaker Diarization Evaluation. In *VI Jornadas en Tecnologías del Habla and II Iberian SLTech Workshop*, pages 397–400, Vigo, Spain, 10-12 November 2010.

[56] M. Diez, M. Penagarikano, A. Varona, L. J. Rodriguez-Fuentes, and G. Bordel. On the use of dot scoring for speaker diarization. In *Iberian Conference on Pattern Recognition and Image Analysis (IbPRIA 2011)*, Las Palmas de Gran Canaria, Spain, 8-10 June 2011.

[57] M. Diez, L. J. Rodriguez-Fuentes, M. Penagarikano, A. Varona, and G. Bordel. Dimensionality reduction of phone log-likelihood ratio features for spoken language recognition. In *Proceedings of Interspeech 2013*, Lyon, France, 25-29 aug. 2013.

[58] M. Diez, L. J. Rodriguez-Fuentes, M. Penagarikano, A. Varona, and G. Bordel. Using phone log-likelihood ratios as features for speaker recognition. In *Proceedings of Interspeech 2013*, Lyon, France, 25-29 aug. 2013.

[59] M. Diez, A. Varona, M. Penagarikano, L. J. Rodriguez-Fuentes, and G. Bordel. On the use of phone log-likelihood ratios as features in spoken language recognition. In *2012 IEEE Workshop on Spoken Language Technology (SLT)*, Miami, Florida, USA., 2-5 dec. 2012.

[60] M. Diez, A. Varona, M. Penagarikano, L. J. Rodriguez-Fuentes, and G. Bordel. Language recognition on Albayzin 2010 LRE using PLLR features. *Procesamiento del Lenguaje Natural*, (51), 2013.

[61] M. Diez, A. Varona, M. Penagarikano, L. J. Rodriguez-Fuentes, and G. Bordel. New insight into the use of phone log-likelihood ratios as features for language recognition. In *Proceedings of Interspeech 2014*, Singapore, 14-18 sep. 2014.

[62] M. Diez, A. Varona, M. Penagarikano, L. J. Rodriguez-Fuentes, and G. Bordel. On the complementarity of phone posterior probabilities for improved speaker recognition. *IEEE Signal Processing Letters*, 21(6):649–652, jun. 2014.

[63] M. Diez, A. Varona, M. Penagarikano, L. J. Rodriguez-Fuentes, and G. Bordel. On the projection of PLLRs for unbounded feature distributions in spoken language recognition. *IEEE Signal Processing Letters*, 21(9):1073–1077, sep 2014.

[64] M. Diez, A. Varona, M. Penagarikano, L. J. Rodriguez-Fuentes, and G. Bordel. Optimizing PLLR features for spoken language recognition. In *proceedings of the22nd International Conference on Pattern Recognition, ICPR 2014*, pages 779–784, Stockholm, Sweden, 24-28 aug 2014.

[65] G. Evermann, H. Chan, M. Gales, T. Hain, X. Liu, D. Mrva, L. Wang, and P. Woodland. Development of the 2003 CU-HTK conversational telephone speech transcription system. In *Acoustics, Speech and Signal Processing (ICASSP), 2004 IEEE International Conference on*, volume 1, pages 249–52, May 2004.

[66] R. A. Fisher. On the mathematical foundations of theoretical statistics. *Philosophical Transactions of the Royal Society of London. Series A, Containing Papers of a Mathematical or Physical Character*, pages 309–368, 1922.

[67] K. Fukunaga. *Introduction to statistical pattern recognition*. Computer Science and Scientific Computing. Academic Press, second edition, 1990.

[68] M. Gales. Maximum likelihood linear transformations for HMM-based speech recognition. *Computer Speech & Language*, 12(2):75 – 98, 1998.

[69] M. Gales, D. Y. Kim, P. Woodland, H. Y. Chan, D. Mrva, R. Sinha, and S. Tranter. Progress in the CU-HTK broadcast news transcription system. *IEEE Transactions on Audio, Speech, and Language Processing*, 14(5):1513–1525, Sept 2006.

[70] J. García, E. Irigoyen, J. A. Elorriaga, N. Garay, E. Zulueta, J. Rubio, C. Vaquero, M. Penagarikano, J. M. López, et al. Intelligent tutoring system to integrate people with Down syndrome into work environments. In *IADAT International Conference on Education*, pages 120–123, Barcelona, Spain, July 2006.

[71] J. L. Gauvain and C. H. Lee. Maximum a posteriori estimation for multivariate gaussian mixture observations of Markov chains. *IEEE Transactions on Speech and Audio Processing*, 2:291–298, 1994.

[72] P. Gopalakrishnan, D. Kanevsky, A. Nadas, and D. Nahamoo. An inequality for rational functions with applications to some statistical estimation problems. *IEEE Transactions on Information Theory*, 37(1):107–113, Jan 1991.

[73] R. Haeb-Umbach and H. Ney. Improvements in beam search for 10000-word continuous-speech recognition. *IEEE Transactions on Speech and Audio Processing*, 2(2):353–356, Apr 1994.

[74] T. Hain, P. Woodland, G. Evermann, and D. Povey. New features in the CU-HTK system for transcription of conversational telephone speech. In *Acoustics, Speech and Signal Processing (ICASSP), 2001 IEEE International Conference on*, volume 1, pages 57–60, 2001.

[75] T. Hain, P. Woodland, T. Niesler, and E. Whittaker. The 1998 HTK system for transcription of conversational telephone speech. In *Acoustics, Speech and Signal Processing (ICASSP), 1999 IEEE International Conference on*, volume 1, pages 57–60, Mar 1999.

[76] F. Harris. On the use of windows for harmonic analysis with the discrete Fourier transform. *Proceedings of the IEEE*, 66(1):51–83, Jan 1978.

[77] H. Hermansky. Perceptual linear predictive (PLP) analysis of speech. *Journal of the Acoustical Society of America*, 87:1738–1752, 1990.

[78] H. Hermansky and N. Morgan. RASTA processing of speech. *IEEE Transactions on Speech and Audio Processing*, 2(4):578–589, Oct. 1994.

[79] X. Huang, A. Acero, and H.-W. Hon. *Spoken Language Processing: A Guide to Theory, Algorithm and System Development*. Prentice Hall PTR, Upper Saddle River, NJ, USA, April 2001.

[80] X. Huang, F. Alleva, H. wuen Hon, M. yuh Hwang, and R. Rosenfeld. The Sphinx-II speech recognition system: An overview. *Computer, Speech and Language*, 7:137–148, 1992.

[81] X. Huang, F. Alleva, M. yuh Hwang, and R. Rosenfeld. An overview of the Sphinx-II speech recognition system. In *In Proceedings of ARPA Human Language Technology Workshop*, pages 81–86, 1993.

[82] J. M. Huerta, S. Chen, and R. M. Stern. The 1998 Carnegie Mellon University Sphinx-3 spanish broadcast news transcription system. In *In Proceedings of the 1999 DARPA Speech Recognition Workshop. DARPA*, 1999.

[83] D. Huggins-Daines, M. Kumar, A. Chan, A. Black, M. Ravishankar, and A. Rudnicky. Pocketsphinx: A free, real-time continuous speech recognition system for hand-held devices. In *Acoustics, Speech and Signal Processing (ICASSP), 2006 IEEE International Conference on*, volume 1, pages I–I, May 2006.

[84] M. J. Hunt. Figures of merit for assessing connected-word recognisers. *Speech Communication*, 9(4):329–336, 1990.

[85] M. Hwamg, R. Rosenfeld, E. Theyer, R. Mosur, L. Chase, R. Weide, X. Huang, and F. Alleva. Improving speech recognition performance via phone-dependent VQ codebooks and adaptive language models in Sphinx-II. In *Acoustics, Speech, and Signal Processing (ICASSP), 1994 IEEE International Conference on*, volume 1, pages 549–552, Apr 1994.

[86] International Organization for Standardization. *ISO 8879:1986: Information processing — Text and office systems — Standard Generalized Markup Language (SGML)*. International Organization for Standardization, Geneva, Switzerland, Aug. 1986.

[87] E. Irigoyen, K. L. de Ipiña, N. Garay, A. Goicoechea, A. Ezeiza, M. Penagarikano, G. Bordel, A. Conde, M. Larrañaga, L. J. Rodriguez-Fuentes, J. López, E. Zulueta, M. Graña, J. L. de Ipiña, J. Rubio, C. Vaquero, and A. Soraluze. Intelligent tutoring system for people with cognitive disabilities. Algarve, Portugal, 2008.

[88] E. Irigoyen, K. López de Ipiña, N. Garay, I. Fajardo, A. Goicoechea, A. Ezeiza, M. Penagarikano, G. Bordel, A. Conde, M. Larrañaga, A. Arruti, L. Rodríguez, J. López, E. Zulueta, M. Graña, and J. Rubio. A robust intelligent system for interacting with people with cognictive disabilities. In *INTED 2008*, Valencia, Spain, 2008.

[89] S. M. Katz. Estimation of probabilities from sparse data for the language model component of a speech recognizer. In *IEEE Transactions on Acoustics, Speech and Signal Processing*, pages 400–401, 1987.

[90] E. Khoury, B. Vesnicer, J. Franco-Pedroso, R. Violato, Z. Boulkenafet, L. M. Fernandez, M. Diez, J. Kosmala, H. Khemiri, T. Cipr, M. G. R. Saeidi, J. Zganec-Gros, R. Z. Candil, F. Simoes, M. Bengherabi, A. A. Marquina, M. Penagarikano, A. Abad, M. Boulayemen, P. Schwarz, D. V. Leeuwen, J. Gonzalez-Domínguez, M. U. Neto, E. Boutellaa, P. G. Vilda, A. Varona, D. Petrovska-Delacretaz, P. Matejka, J. Gonzalez-Rodríguez, T. Pereira, F. Harizi, L. J. Rodriguez-Fuentes, L. E. Shafey, M. Angeloni, G. Bordel, G. Chollet, and S. Marcel. The 2013 speaker recognition evaluation in mobile environment. In *The 6th IAPR International Conference on Biometrics (ICB-2013)*, Madrid, Spain., June 4 - 7 2013.

[91] D. Kim, H. Chan, G. Evermann, M. Gales, D. Mrva, K. Sim, and P. Woodland. Development of the CU-HTK 2004 broadcast news transcription systems. In *Acoustics, Speech and Signal Processing (ICASSP), 2005 IEEE International Conference on*, volume 1, pages 861–864, March 2005.

[92] D. Kim, G. Evermann, T. Hain, D. Mrva, S. Tranter, L. Wang, and P. Woodland. Recent advances in broadcast news transcription. In *Automatic Speech Recognition and Understanding, ASRU. 2003 IEEE Workshop on*, pages 105–110, Nov 2003.

[93] T. Kinnunen, R. Saeidi, F. Sedlak, K. A. Lee, J. Sandberg, M. Hansson-Sandsten, and H. Li. Low-variance multitaper MFCC features: A case study in robust speaker verification. *IEEE Transactions on Audio, Speech, and Language Processing*, 20(7):1990–2001, 2012.

[94] M. Kohler and M. Kennedy. Language identification using shifted delta cepstra. In *Circuits and Systems, 2002. MWSCAS-2002. The 2002 45th Midwest Symposium on*, volume 3, pages 69–72, Aug. 2002.

[95] P. Lamere, P. Kwok, W. Walker, E. Gouvea, R. Singh, B. Raj, and P. Wolf. Design of the CMU Sphinx-4 decoder. In *Proceedings of Eurospeech*, Geneva, Switzerland, 2003. ISCA.

[96] A. Lee, T. Kawahara, and K. Shikano. Julius - an open source real-time large vocabulary recognition engine. In P. Dalsgaard, B. Lindberg, H. Benner, and Z.-H. Tan, editors, *Proceedings of Interspeech 2001*, pages 1691–1694. ISCA, 2001.

[97] B. T. Lee, L. Masinter, and M. Mccahill. RFC 1738: Uniform Resource Locator (URL). http://tools.ietf.org/html/rfc1738, 1994. [Online; accessed 02-Dec-2015].

[98] K. Lee. *Automatic Speech Recognition: The Development of the Sphinx System.* Kluwer Academic Publishers, Dordrecht, Netherlands, 1989.

[99] K.-F. Lee. *Large-vocabulary Speaker-independent Continuous Speech Recognition: The Sphinx System.* PhD thesis, Pittsburgh, PA, USA, 1988. AAI8826533.

[100] K.-F. Lee, H.-W. Hon, M.-Y. Hwang, S. Mahajan, and R. Reddy. The Sphinx speech recognition system. In *Acoustics, Speech and Signal Processing (ICASSP), 1989 IEEE International Conference on*, volume 1, pages 445–448, May 1989.

[101] K.-F. Lee, H.-W. Hon, and R. Reddy. An overview of the Sphinx speech recognition system. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 38(1):35–45, Jan 1990.

[102] L. Lee and R. Rose. Speaker normalization using efficient frequency warping procedures. In *Acoustics, Speech and Signal Processing (ICASSP), 1996 IEEE International Conference on*, volume 1, pages 353–356, May 1996.

[103] C. Leggetter and P. Woodland. Maximum likelihood linear regression for speaker adaptation of continuous density hidden Markov models. *Computer Speech & Language*, 9(2):171 – 185, 1995.

[104] Y. Linde, A. Buzo, and R. Gray. An algorithm for vector quantizer design. *IEEE Transactions on Communications*, 28(1):84–95, Jan 1980.

[105] K. López de Ipiña, N. Ezeiza, G. Bordel, M. Graña, and M. Penagarikano. Selection of lexical units for continuous speech recognition of Basque. In *Proccedings of the International Symposium on Social Communication*, 2003.

[106] K. López de Ipiña, I. Torres, L. Oñederra, A. Varona, N. Ezeiza, M. Penagarikano, M. Hernandez, and L. J. Rodriguez-Fuentes. First approach to the selection of lexical units for continuous speech recognition of Basque. In *Proceedings of the International Conference on Speech and Language Processing (ICSLP)*, volume 2, pages 531–534, Beijing, China, October 2000.

[107] K. López de Ipiña, E. Zulueta, M. Penagarikano, G. Bordel, N. Garay, J. Elorriaga, J. Lopez, E. Irigoyen, A. Ezeiza, J. López de Ipiña, J. Rubio, C. Vaquero, B. Rubio, R. Molinero, and J. Aguirre. Sistema tutor inteligente (STI) para la integración laboral de trabajadores con síndrome de Down. In *VI Congreso de Interacción Persona Ordenador*, Granada, Spain, Septiembre 2005.

[108] J. Macías. *Arquitecturas y Métodos en Sistemas de Reconocimiento Automático de Habla de Gran Vocabulario*. PhD thesis, Universidad Politécnica de Madrid, 2001.

[109] A. Marzal and E. Vidal. Computation of normalized edit distance and applications. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(9):926–932, Sep 1993.

[110] G. McLachlan and T. Krishnan. *The EM algorithm and extensions*. Wiley, New York, 1997.

[111] B. Merialdo. Phonetic recognition using hidden Markov models and maximum mutual information training. In *Acoustics, Speech, and Signal Processing (ICASSP), 1988 IEEE International Conference on*, volume 1, pages 111–114, Apr 1988.

[112] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM.

[113] H. Ney, R. Haeb-Umbach, B.-H. Tran, and M. Oerder. Improvements in beam search for 10000-word continuous speech recognition. *Acoustics, Speech, and Signal Processing, IEEE International Conference on*, 1:9–12, 1992.

[114] I. Odriozola, I. Hernaez, M. Torres, L. J. Rodriguez-Fuentes, M. Penagarikano, and E. Navas. Basque Speecon-like and Basque SpeechDat MDB-600: speech databases for the development of ASR technology for Basque. In *proceedings of the 9th international conference on language resources and evaluation (lrec'14)*, Reykjavik, Iceland, 26-31 may 2014. European Language Resources Association (ELRA).

[115] D. Panchenko. 18.443 - Statistics for Applications, Fall 2003. *Massachusetts Institute of Technology: MIT OpenCourseWare*, 2003. [Online; accessed 02-Dec-2015. License: Creative Commons BY-NC-SA].

[116] S. H. Parfitt and R. A. Sharman. A bi-directional model of English pronunciation. In *Proceedings of Eurospeech*. ISCA, 1991.

[117] G. Patanè and M. Russo. The enhanced LBG algorithm. *Neural Networks*, 14(9):1219–1237, 2001.

[118] B. Payette. *Windows PowerShell in Action*. Manning Publications Co., 2 edition, 2007.

[119] J. Pelecanos and S. Sridharan. Feature Warping for Robust Speaker Verification. In *Odyssey 2011: the speaker and language recognition workshop*, pages 213–218, 2001.

[120] M. Penagarikano and G. Bordel. Speech-to-text translation by a non-word lexical unit based system. In *Proceedings of the ISSPA'99*, volume 1, pages 111–114, Brisbane, Australia, August 1999.

[121] M. Penagarikano and G. Bordel. Layered Markov Models: A new architectural approach to automatic speech recognition. In *Proceedings of the MLSP Workshop*, pages 305–314, Sao Luis, Brasil, October 2004.

[122] M. Penagarikano and G. Bordel. Sautrela: A highly modular open source speech recognition framework. In *Proceedings of the ASRU Workshop*, pages 386–391, San Juan, Puerto Rico, December 2005.

[123] M. Penagarikano, G. Bordel, S. Bilbao, M. Zamalloa, and L. J. Rodriguez-Fuentes. Sautrela: un entorno de desarrollo versátil para las tecnologías del habla. In *Actas de las V Jornadas en Tecnología del Habla*, pages 233–235, Bilbao, November 2008.

[124] M. Penagarikano, G. Bordel, and L. J. Rodriguez-Fuentes. Unified training of WFSA through a generic interface. In *Proceedings of Spoken Language Technology Workshop, 2006. IEEE*, pages 122–125, Palm Beach, Aruba, December 2006.

[125] M. Penagarikano, G. Bordel, L. J. Rodriguez-Fuentes, and J. Uribe. University of the Basque Country + Ikerlan system for NIST 2007 Language Recognition Evaluation. In *2007 NIST Language Recognition Evaluation (LRE) Workshop*, Orlando, Florida, USA, 2007.

[126] M. Penagarikano, G. Bordel, L. J. Rodriguez-Fuentes, and M. Zamalloa. Diseño e implementación de una interfaz genérica para la estimación de autómatas de estados finitos ponderados. In *IV Jornadas en Tecnología del Habla*, pages 373–377, Zaragoza, Spain, November 2006.

[127] M. Penagarikano, G. Bordel, A. Varona, and K. López de Ipiña. Using non-word lexical units in automatic speech understanding. In *Acoustics, Speech and Signal Processing (ICASSP), 1999 IEEE International Conference on*, volume 2, pages 621–624, Phoenix, USA, May 1999.

[128] M. Penagarikano, M. Diez, L. J. Rodriguez-Fuentes, A. Varona, and G. Bordel. University of the Basque Country systems for the NIST i-vector Machine Learning Challenge. In *Speaker Odyssey 2014*, Joensuu, Finland, 16-19 jun. 2014.

[129] M. Penagarikano, A. Varona, M. Diez, L. J. Rodriguez-Fuentes, and G. Bordel. A speaker recognition system based on sufficient-statistics-space channel-compensation and dot-scoring. In *VI Jornadas en Tecnologías del Habla and II Iberian SLTech Workshop*, pages 135–138, Vigo, Spain, 10-12 November 2010.

[130] M. Penagarikano, A. Varona, M. Diez, L. J. Rodriguez-Fuentes, and G. Bordel. University of the Basque Country system for NIST 2010 Speaker Recognition Evaluation. In *V Jornadas de Reconocimiento Biométrico de Personas*, Huesca, Spain, 2-3 September 2010.

[131] M. Penagarikano, A. Varona, M. Diez, L. J. Rodriguez-Fuentes, and G. Bordel. University of the Basque Country system for NIST 2010 Speaker Recognition Evaluation. In *2010 NIST Speaker Recognition Evaluation (SRE) Workshop*, Brno, Czech Republic, 24-25 June 2010.

[132] M. Penagarikano, A. Varona, M. Diez, L. J. Rodriguez-Fuentes, and G. Bordel. Study of different backends in a state-of-the-art language recognition system. In *Proceedings of Interspeech 2012*, Portland, Oregon, USA, 9-13 sept. 2012.

[133] M. Penagarikano, A. Varona, L. J. Rodriguez-Fuentes, and G. Bordel. Improved modeling of cross-decoder phone co-occurrences in SVM-based phonotactic language recognition. In *Odyssey 2010: The Speaker and Language Recognition Workshop*, Brno, Czech Republic, 28 June - 1 July 2010.

[134] M. Penagarikano, A. Varona, L. J. Rodriguez-Fuentes, and G. Bordel. Using cross-decoder co-occurrences of phone n-grams in SVM-based phonotactic language recognition. In *Proceedings of Interspeech 2010*, Makuhari, Japan, 26-30 September 2010.

[135] M. Penagarikano, A. Varona, L. J. Rodriguez-Fuentes, and G. Bordel. Using cross-decoder phone coocurrences in phonotactic language recognition. In *Acoustics, Speech and Signal Processing (ICASSP), 2010 IEEE International Conference on*, Dallas(Texas), USA, March 2010.

[136] M. Penagarikano, A. Varona, L. J. Rodriguez-Fuentes, and G. Bordel. Dimensionality reduction for using high-order n-grams in SVM-based phonotactic language recognition. In *Proceedings of Interspeech 2011*, Florence, Italy, 28-31 August 2011.

[137] M. Penagarikano, A. Varona, L. J. Rodriguez-Fuentes, and G. Bordel. A dynamic approach to the selection of high order n-grams in phonotactic language recognition. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, Prague, Czech Republic, 22-27 May 2011.

[138] M. Penagarikano, A. Varona, L. J. Rodriguez-Fuentes, and G. Bordel. Improved modeling of cross-decoder phone co-occurrences in SVM-based phonotactic language recognition. *IEEE Transactions on Audio, Speech, and Language Processing*, 19(8):2348–2363, Nov. 2011.

[139] M. Penagarikano, A. Varona, L. J. Rodriguez-Fuentes, M. Díez, and G. Bordel. University of the Basque Country (EHU) systems for the 2011 NIST Language Recognition Evaluation. In *Proceedings of the NIST 2011 Language Recognition Evaluation (LRE) Workshop*, Atlanta (USA), 6-7 december 2011.

[140] M. Penagarikano, A. Varona, L. J. Rodriguez-Fuentes, M. Diez, and G. Bordel. The EHU systems for the NIST 2011 Language Recognition Evaluation. In *Proceedings of Interspeech 2012*, Portland, Oregon, USA, 9-13 sept. 2012.

[141] M. Penagarikano, A. Varona, M. Zamalloa, L. J. Rodriguez-Fuentes, G. Bordel, and J. Uribe. University of the Basque Country + Ikerlan system for NIST 2009 Language Recognition Evaluation. In *2009 NIST Language Recognition Evaluation (LRE) Workshop*, Baltimore, MD, USA, 2009.

[142] P. Placeway, S. Chen, M. Eskenazi, U. Jain, V. Parikh, B. Raj, M. Ravishankar, R. Rosenfeld, K. Seymore, M. Siegler, R. Stern, and E. Thayer. The 1996 Hub-4 Sphinx-3 system. In *In Proceedings of the 1996 DARPA Speech Recognition Workshop. DARPA*. The, 1996.

[143] D. Povey, A. Ghoshal, G. Boulianne, L. Burget, O. Glembek, N. Goel, M. Hannemann, P. Motlicek, Y. Qian, P. Schwarz, J. Silovsky, G. Stemmer, and K. Vesely. The Kaldi speech recognition toolkit. In *IEEE 2011 Workshop on Automatic Speech Recognition and Understanding*, Hawaii, US, Dec. 2011.

[144] F. Prat, P. Aibar, A. Marzal, and E. Vidal. El problema de la evaluación de un sistema de reconocimiento automático del habla mediante un único valor numérico. Technical report, DSIC II/15/94, UPV, 1994.

[145] L. R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. In *Proceedings of IEEE*, volume 77. IEEE, February 1988.

[146] M. Ravishankar, R. Singh, B. Raj, and R. M. Stern. The 1999 CMU 10x real time broadcast news transcription system. In *In Proceedings of the 2000 DARPA workshop on Automatic Transcription of Broadcast News. DARPA*, 2000.

[147] D. A. Reynolds, T. F. Quatieri, and R. B. Dunn. Speaker verification using adapted Gaussian mixture models. *Digital Signal Processing*, 10:19–41, January 2000.

[148] K. Riedel and A. Sidorenko. Minimum bias multiple taper spectral estimation. *IEEE Transactions on Signal Processing*, 43(1):188–195, 1995.

[149] L. J. Rodriguez-Fuentes. *Estudio y modelización acústica del habla espontánea en diálogos hombre-máquina y entre personas.* PhD thesis, Departamento de Electricidad y Electrónica, Facultad de Ciencia y Tecnología, Universidad del País Vasco, Leioa, Spain, Julio 2004.

[150] L. J. Rodriguez-Fuentes, N. Brümmer, M. Penagarikano, A. Varona, G. Bordel, and M. Diez. The Albayzin 2012 Language Recognition Evaluation. In *Proceedings of Interspeech 2013*, Lyon, France, 25-29 aug. 2013.

[151] L. J. Rodriguez-Fuentes, N. Brümmer, M. Penagarikano, A. Varona, M. Diez, and G. Bordel. The Albayzin 2012 Language Recognition Evaluation plan. In *Iberspeech 2012*, Madrid, Spain, 21-23 nov. 2012.

[152] L. J. Rodriguez-Fuentes and M. Penagarikano. Mediaeval 2013 spoken web search task: System performance measures. Technical Report TR-2013-1, Department of Electricity and Electronics, University of the Basque Country, 2013.

[153] L. J. Rodriguez-Fuentes, M. Penagarikano, and G. Bordel. *A Simple but Effective Approach to Speaker Tracking in Broadcast News*, volume LCNS 4478 of *Lecture Notes in Computer Science*, pages 48–55. Pattern Recognition and Image Analysis (IbPRIA 2007), Joan Martí, José Miguel Benedí, Ana Maria Mendonça and Joan Serrat (Eds.), Springer Verlag, Berlin Heidelberg, 2007.

[154] L. J. Rodriguez-Fuentes, M. Penagarikano, G. Bordel, and A. Varona. The Albayzin 2008 Language Recognition Evaluation. In *Odyssey 2010: The Speaker and Language Recognition Workshop*, Brno, Czech Republic, 28 June - 1 July 2010.

[155] L. J. Rodriguez-Fuentes, M. Penagarikano, G. Bordel, A. Varona, and M. Diez. Kalaka: A TV broadcast speech database for the evaluation of language recognition systems. In *7th International Conference on Language Resources and Evaluation*, Valleta, Malta, 17-23 May 2010.

[156] L. J. Rodriguez-Fuentes, M. Penagarikano, A. Varona, M. Díez, and G. Bordel. The Albayzin 2010 Language Recognition Evaluation. In *Proceedings of Interspeech 2011*, Florence, Italy, 28-31 August 2011.

[157] L. J. Rodriguez-Fuentes, M. Penagarikano, A. Varona, M. Diez, and G. Bordel. GTTS systems for the Albayzin 2010 Audio Segmentation Evaluation. In *VI Jornadas en Tecnologías del Habla and II Iberian SLTech Workshop*, pages 419–420, Vigo, Spain, 10-12 November 2010.

[158] L. J. Rodriguez-Fuentes, M. Penagarikano, A. Varona, M. Diez, and G. Bordel. Overview of the Albayzin 2010 Language Recognition Evaluation: database design, evaluation plan and preliminary analysis of results. In *VI Jornadas en Tecnologías del Habla and II Iberian SLTech Workshop*, pages 309–316, Vigo, Spain, 10-12 November 2010.

[159] L. J. Rodriguez-Fuentes, M. Penagarikano, A. Varona, M. Diez, and G. Bordel. Kalaka-2: a TV broadcast speech database for the recognition of Iberian languages in clean and noisy environments. In *Proceedings of the eighth International Conference on Language Resources and Evaluation (LREC'12)*, pages 99–105, Istambul, Turkey, 23-25 may 2012. European Language Resources Association (ELRA).

[160] L. J. Rodriguez-Fuentes, M. Penagarikano, A. Varona, M. Diez, and G. Bordel. Kalaka-3: a database for the recognition of spoken European languages on

YouTube audios. In *Proceedings of the 9th International Conference on Language Resources and Evaluation (LREC'14)*, Reykjavik, Iceland, 26-31 may 2014. European Language Resources Association (ELRA).

[161] L. J. Rodriguez-Fuentes, M. Penagarikano, A. Varona, M. Diez, and G. Bordel. Kalaka-3: A database for the assessment of spoken language recognition technology on YouTube audios. *Language Resources and Evaluation*, In press 2016.

[162] L. J. Rodriguez-Fuentes, M. Penagarikano, A. Varona, M. Diez, G. Bordel, A. Abad, D. Martinez, J. Villalba, A. Ortega, and E. Lleida. The BLZ system for the 2011 NIST Language Recognition Evaluation. In *NIST 2011 Language Recognition Evaluation Workshop*, Atlanta (USA), 6-7 december 2011.

[163] L. J. Rodriguez-Fuentes, M. Penagarikano, A. Varona, M. Diez, G. Bordel, A. Abad, D. Martinez, J. Villalba, A. Ortega, and E. Lleida. The BLZ submission to the NIST 2011 LRE: data collection, system development and performance. In *Proceedings of Interspeech 2012*, Portland, Oregon, USA, 9-13 sept. 2012.

[164] L. J. Rodriguez-Fuentes, M. Penagarikano, A. Varona, M. Diez, G. Bordel, D. Martinez, J. Villalba, A. Miguel, A. Ortega, E. Lleida, A. Abad, O. Koller, I. Trancoso, P. Lopez-Otero, L. Docio-Fernandez, C. Garcia-Mateo, R. Saeidi, M. Soufifar, T. Kinnunen, T. Svendsen, and F. P. Multi-site heterogeneous system fusions for the Albayzin 2010 Language Recognition Evaluation. In *IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU'11)*, Hawaii (USA), 12-15 december 2011.

[165] L. J. Rodriguez-Fuentes, I. Torres, J. Alcaide, A. Varona, K. López de Ipiña, M. Penagarikano, and G. Bordel. An integrated system for spanish CSR tasks. In *Proceedings of Eurospeech*, volume 2, pages 951–954, Budapest, Hungary, September 1999.

[166] L. J. Rodriguez-Fuentes, I. Torres, J. Alcaide, A. Varona, K. López de Ipiña, M. Penagarikano, and G. Bordel. A new integrated system for the continuous speech recognition of Spanish. In *Proccedings of the VIII Symposium Nacional de Reconocimiento de Formas y Análisis de Imágenes*, 1999.

[167] L. J. Rodriguez-Fuentes, A. Varona, M. Diez, M. Penagarikano, and G. Bordel. Evaluation of technology using broadcast speech: performance and challenges. In *Odyssey 2012: the speaker and language recognition workshop*, Singapore, 25-28 june 2012.

[168] L. J. Rodriguez-Fuentes, A. Varona, M. Penagarikano, G. Bordel, and M. Diez. GTTS systems for the SWS task at MediaEval 2013. In *Proceedings of the MediaEval 2013 Multimedia Benchmark Workshop*, Barcelona, 18-19 oct. 2013.

[169] L. J. Rodriguez-Fuentes, A. Varona, M. Penagarikano, G. Bordel, and M. Diez. GTTS-EHU systems for QUESST at MediaEval 2014. In *Proceedings of the MediaEval 2012 Multimedia Benchmark Workshop*, volume 1263, Barcelon, Spain, October 16-17 2014.

[170] L. J. Rodriguez-Fuentes, A. Varona, M. Penagarikano, G. Bordel, and M. Diez. High-performance query-by-example spoken term detection on the SWS 2013 evaluation. In *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on)*, Florence, Italy, 4-9 May 2014.

[171] L. J. Rodriguez-Fuentes, A. Varona, M. Penagarikano, M. Diez, and G. Bordel. Spoken language recognition in conversational telephone speech and TV broadcast news (GLOSA). In *XXVI Congreso de la Sociedad Española para el Procesamiento de Lenguaje Natural (SEPLN)*, Huelva, Spain, 5-7 September 2011.

[172] J. Rubio, C. Vaquero, J. M. L. de Ipiña, E. Irigoyen, K. L. de Ipiña, N. Garay, A. Conde, M. Larrañaga, A. Ezeiza, A. Soraluze, M. Penagarikano, G. Bordel, L. J. Rodriguez-Fuentes, J. M. López, M. Ezquerra, and D. Oregi. *Tutor Project: An Intelligent Tutoring System to Improve Cognitive Disabled People Integration*, pages 729–732. Number LNCS 5105 in Lecture Notes in Computer Science. Springer-Verlag, Linz, Austria, July 2008.

[173] D. Rybach, C. Gollan, G. Heigold, B. Hoffmeister, J. Lööf, R. Schlüter, and H. Ney. The RWTH Aachen University open source speech recognition system. In *Proceedings of Interspeech 2009*, pages 2111–2114, Brighton, UK, 6-10 sep. 2009. ISCA.

[174] E. Segarra. *Una Aproximación Inductiva a la Comprensión del Discurso Continuo*. PhD thesis, Universidad Politécnica de Valencia, 1993.

[175] A. Sixtus, S. Molau, S. Kanthak, R. Schluter, and H. Ney. Recent improvements of the rwth large vocabulary speech recognition system on spontaneous speech. In *Acoustics, Speech and Signal Processing (ICASSP), 2000 IEEE International Conference on*, volume 3, pages 1671–1674, 2000.

[176] The Unicode Consortium. The Unicode Standard. Technical Report Version 6.0.0, Unicode Consortium, Mountain View, CA, 2011.

[177] M. Thompson, D. Farley, M. Barker, P. Gee, and A. Stewart. LMAX Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads. Technical report, Department of Engineering, Cambridge University, December 2011.

[178] P. A. Torres-carrasquillo, E. Singer, M. A. Kohler, and J. R. Deller. Approaches to language identification using Gaussian mixture models and shifted delta cepstral features. In *Proc. ICSLP 2002*, pages 89–92, 2002.

[179] A. Varona. *Modelos k-explorables en sentido estricto integrados en un sistema de reconocimiento automático del habla*. PhD thesis, Departamento de Electricidad y Electrónica, Facultad de Ciencia y Tecnología, Universidad del País Vasco, Leioa, Spain, Abril 2000.

[180] A. Varona, S. Nieto, L. J. Rodriguez-Fuentes, M. Penagarikano, G. Bordel, and M. Diez. A spoken document retrieval system for TV broadcast news in Spanish

and Basque. In *XXVI Congreso de la Sociedad Española para el Procesamiento de Lenguaje Natural (SEPLN)*, Huelva, Spain, 5-7 September 2011.

[181] A. Varona, M. Penagarikano, L. J. Rodriguez-Fuentes, and G. Bordel. On the use of lattices of time-synchronous cross-decoder phone co-occurrences in a SVM-phonotactic language recognition system. In *Proceedings of Interspeech 2011*, Florence, Italy, 28-31 August 2011.

[182] A. Varona, M. Penagarikano, L. J. Rodriguez-Fuentes, G. Bordel, and M. Diez. GTTS system for the spoken web search task at MediaEval 2012. In *Proceedings of the MediaEval 2012 Multimedia Benchmark Workshop*, volume 927, Pisa, Italy, October 4-5 2012.

[183] A. Varona, M. Penagarikano, L. J. Rodriguez-Fuentes, G. Bordel, and M. Diez. GTTS systems for the query-by-example spoken term detection task of the Albayzin 2012 Search on Speech Evaluation. pages 619–625, Madrid, Spain, november 21-23 2012.

[184] A. Varona, M. Penagarikano, L. J. Rodriguez-Fuentes, G. Bordel, and M. Diez. Using time-synchronous phone co-occurrences in a SVM-phonotactic dialect recognition system. In *Proceedings of Interspeech 2012*, Portland, Oregon, USA, 9-13 sept. 2012.

[185] A. Varona, M. Penagarikano, L. J. Rodriguez-Fuentes, M. Diez, and G. Bordel. Verification of the four Spanish official languages on TV show recordings. In *XXV Congreso de la Sociedad Española para el Procesamiento de Lenguaje Natural (SEPLN)*, Valencia, Spain, 8-10 September 2010.

[186] A. Varona, L. J. Rodriguez-Fuentes, M. Penagarikano, S. Nieto, M. Diez, and G. Bordel. Search and access to information contained in the speech of multimedia resources. In *XXV Congreso de la Sociedad Española para el Procesamiento de Lenguaje Natural (SEPLN)*, Valencia, Spain, 8-10 September 2010.

[187] E. Vidal, A. Marzal, and P. Aibar. Fast computation of normalized edit distances. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17(9):899–902, Sep 1995.

[188] A. J. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, IT-13(2):260–269, April 1967.

[189] J. Volkmann, S. S. Stevens, and E. B. Newman. A scale for the measurement of the psychological magnitude pitch. *The Journal of the Acoustical Society of America*, 8(3):208–208, 1937.

[190] W. Walker, P. Lamere, P. Kwok, B. Raj, R. Singh, E. Gouvea, P. Wolf, and J. Woelfel. Sphinx-4: A flexible open source framework for speech recognition. Technical Report TR-2004-139, Sun Microsystems, 2004.

[191] Wikipedia. Arpabet — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Arpabet. [Online; accessed 02-Dec-2015].

[192] Wikipedia. Beta function — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Beta_function. [Online; accessed 02-Dec-2015].

[193] Wikipedia. Categorical distribution — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Categorical_distribution. [Online; accessed 02-Dec-2015].

[194] Wikipedia. Combination — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Combination. [Online; accessed 02-Dec-2015].

[195] Wikipedia. Conjugate prior — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Conjugate_prior. [Online; accessed 02-Dec-2015].

[196] Wikipedia. Cross entropy — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Cross_entropy. [Online; accessed 02-Dec-2015].

[197] Wikipedia. Dirichlet distribution — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Dirichlet_distribution. [Online; accessed 02-Dec-2015].

[198] Wikipedia. Discrete cosine transform — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Discrete_cosine_transform. [Online; accessed 02-Dec-2015].

[199] Wikipedia. Expectation-maximization algorithm — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Expectation-maximization_algorithm. [Online; accessed 02-Dec-2015].

[200] Wikipedia. Exponential family — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Exponential_family. [Online; accessed 02-Dec-2015].

[201] Wikipedia. Inverse-wishart distribution — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Inverse-Wishart_distribution. [Online; accessed 02-Dec-2015].

[202] Wikipedia. Multivariate gamma function — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Multivariate_gamma_function. [Online; accessed 02-Dec-2015].

[203] Wikipedia. Named pipe — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Named_pipe. [Online; accessed 02-Dec-2015].

[204] Wikipedia. Normal-inverse-wishart distribution — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Normal-inverse-Wishart_distribution. [Online; accessed 02-Dec-2015].

[205] Wikipedia. Perplexity — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Perplexity. [Online; accessed 02-Dec-2015].

[206] Wikipedia. Plate notation — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Plate_notation. [Online; accessed 02-Dec-2015].

[207] Wikipedia. Process substitution — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Process_substitution. [Online; accessed 02-Dec-2015].

[208] Wikipedia. Regular expression — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Regular_expression. [Online; accessed 02-Dec-2015].

[209] Wikipedia. Serialization — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Serialization. [Online; accessed 02-Dec-2015].

[210] Wikipedia. Standard generalized markup language — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Standard_Generalized_Markup_Language. [Online; accessed 02-Dec-2015].

[211] Wikipedia. Sufficient statistic — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Sufficient_statistic. [Online; accessed 02-Dec-2015].

[212] Wikipedia. Unicode — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Unicode. [Online; accessed 02-Dec-2015].

[213] Wikipedia. Uniform resource locator — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Uniform_resource_locator. [Online; accessed 02-Dec-2015].

[214] Wikipedia. Unit testing — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Unit_testing. [Online; accessed 02-Dec-2015].

[215] Wikipedia. Window function — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Window_function. [Online; accessed 02-Dec-2015].

[216] Wikipedia. XML — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/XML. [Online; accessed 02-Dec-2015].

[217] Wikipedia. Zip file format — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/ZIP_file_format. [Online; accessed 02-Dec-2015].

[218] L. Wood, D. J. B. Pearce, and F. Novello. Improved vocabulary-independent subword HMM modelling. In *Acoustics, Speech, and Signal Processing (ICASSP), 1991 IEEE International Conference on*, volume 1, pages 181–184, Apr 1991.

[219] P. Woodland. An overview of HTK v3.5. Technical report. Phil Woodland's keynote talk on HTK v3.5 at the fourth meeting of the UK and Irish speech science and technology research community, University of East Anglia.

[220] P. Woodland, M. Gales, D. Pye, and S. Young. Broadcast news transcription using HTK. In *Acoustics, Speech and Signal Processing (ICASSP), 1997 IEEE International Conference on*, volume 2, pages 719–722, Apr 1997.

[221] P. Woodland, J. Odell, V. Valtchev, and S. Young. Large vocabulary continuous speech recognition using HTK. In *Acoustics, Speech, and Signal Processing (ICASSP), 1994 IEEE International Conference on*, volume 2, pages 125–128, Apr 1994.

[222] P. C. Woodland. Speaker adaptation for continuous density HMMs: A review. In *ITRW on Adaptation Methods for Speech Recognition*, pages 11–19, Aug. 2001.

[223] P. C. Woodland and S. J. Young. The HTK tied-state continuous speech recogniser. In *Proceedings of Eurospeech*, Berlin, 1993.

[224] B. Xiang, U. V. Chaudhari, J. Navratil, G. N. Ramaswamy, and R. A. Gopinath. Short-time gaussianization for robust speaker verification. In *Acoustics, Speech and Signal Processing (ICASSP), 2002 IEEE International Conference on*, pages 681–684. IEEE, 2002.

[225] S. Young. The HTK hidden Markov model toolkit: Design and philosophy. *Entropic Cambridge Research Laboratory, Ltd*, 2:2–44, 1994.

[226] S. Young. ATK: An application toolkit for HTK. Technical report, Department of Engineering, Cambridge University, 2007.

[227] S. J. Young, G. Evermann, M. J. F. Gales, T. Hain, D. Kershaw, G. Moore, J. Odell, D. Ollason, D. Povey, V. Valtchev, and P. C. Woodland. *The HTK Book, version 3.4*. Cambridge University Engineering Department, Cambridge, UK, 2006.

[228] M. Zamalloa, G. Bordel, L. J. Rodriguez-Fuentes, and M. Penagarikano. Feature selection based on genetic algorithms for speaker recognition. In *Proceedings of the IEEE Odyssey - The Speaker and Language Recognition Workshop*, Puerto Rico, June 2006.

[229] M. Zamalloa, G. Bordel, L. J. Rodriguez-Fuentes, M. Penagarikano, and J. Uribe. Selección y pesado de parámetros acústicos mediante algoritmos genéticos para el reconocimiento del locutor. In *IV Jornadas en Tecnologia del Habla*, pages 349–354, Zaragoza, Spain, November 2006.

[230] M. Zamalloa, G. Bordel, L. J. Rodriguez-Fuentes, M. Penagarikano, and J. Uribe. Using genetic algorithms to weight acoustic features for speaker recognition. In *Proceedings of the INTERSPEECH - ICSLP*, Pittsburgh, September 2006.

[231] M. Zamalloa, M. Penagarikano, L. J. Rodriguez-Fuentes, G. Bordel, and J. Uribe. An online speaker tracking system for ambient intelligence environments. In *2nd International Conference on Agents and Artificial Intelligence (ICAART)*, Valencia, Spain, January 2010.

[232] M. Zamalloa, L. J. Rodriguez-Fuentes, G. Bordel, M. Penagarikano, J. Parra, A. Uribarren, and J. Uribe. Low-latency speaker tracking and SOA-compliant services for ambient intelligence environments. In *VI Jornadas en Tecnologías del Habla and II Iberian SLTech Workshop*, pages 157–160, Vigo, Spain, 10-12 November 2010.

[233] M. Zamalloa, L. J. Rodriguez-Fuentes, G. Bordel, M. Penagarikano, and J. Uribe. Low-latency online speaker tracking on the AMI corpus of meeting conversations. In *Acoustics, Speech and Signal Processing (ICASSP), 2010 IEEE International Conference on*, Dallas(Texas), USA, March 2010.

[234] M. Zamalloa, L. J. Rodriguez-Fuentes, M. Penagarikano, G. Bordel, and J. Uribe. Comparing genetic algorithms to principal component analysis and linear discriminant análisis in reducing feature dimensionality for speaker recognition. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, Atlanta, USA, July 2008.

[235] M. Zamalloa, L. J. Rodriguez-Fuentes, M. Penagarikano, G. Bordel, and J. Uribe. Feature dimensionality reduction through genetic algorithms for faster speaker recognition. In *Proceedings of the 16th European Signal Processing Conference (EUSIPCO)*, Laussane, Switzerland, August 2008.

[236] M. Zamalloa, L. J. Rodriguez-Fuentes, M. Penagarikano, G. Bordel, and J. Uribe. Feature selection vs. feature transformation in reducing dimensionality for speaker recognition. In *Actas de las V Jornadas en Tecnología del Habla*, pages 45–48, Bilbao, November 2008.

[237] M. Zamalloa, L. J. Rodriguez-Fuentes, M. Penagarikano, G. Bordel, and J. Uribe. Improving robustness in open set speaker identification by shallow source modelling. In *Odyssey 2008: The Speaker and Language Recognition Workshop*, Stellenbosch, South Africa, January 2008.

[238] M. Zamalloa, L. J. Rodriguez-Fuentes, M. Penagarikano, G. Bordel, and J. Uribe. Increasing robustness to acoustically uncovered signals in speaker verification through shallow source modelling. In *Proceedings of the 16th European Signal Processing Conference (EUSIPCO)*, Laussane, Switzerland, August 2008.

[239] M. Zamalloa, L. J. Rodriguez-Fuentes, M. Penagarikano, G. Bordel, and J. Uribe. University of the Basque Country + Ikerlan system for NIST 2008 Speaker Recognition Evaluation. In *2008 NIST Speaker Recognition Evaluation (SRE) Workshop*, Montreal, Canada, 2008.

[240] C. Zhang and P. C. Woodland. A general artificial neural network extension for HTK. In *Proceedings of Interspeech 2015*, pages 3581–3585, Dresden, Germany, 6-10 September 2015.

[241] M. Zissman. Comparison of four approaches to automatic language identification of telephone speech. *IEEE Transactions on Speech and Audio Processing*, 4(1):31–, Jan 1996.

Índice alfabético