universidad del país vasco euskal herriko unibertsitatea

Facultad de Informática / Informatika Fakultatea

Integración del algoritmo CHAID y una adaptación de éste, CHAID*, en la plataforma Weka

Alumno/a: D. Oscar Teixeira Martín

Director/a: D. Jesus María Pérez de la Fuente

Proyecto Fin de Carrera, junio de 2016

RESUMEN

El presente documento recoge la memoria de un proyecto de fin de carrera para la titulación de Ingeniería en Informática de la Facultad de Informática de Donostia-San Sebastián. El proyecto se enmarca dentro del área de trabajo del grupo de investigación ALDAPA (http://www.aldapa.eus/), centrado en el área de la minería de datos. Concretamente, se centra en una de sus líneas de investigación dedicada a los “algoritmos de aprendizaje supervisado con capacidades explicativas”.

Este proyecto tiene dos objetivos principales: en primer lugar, integrar el algoritmo CHAID en una plataforma de software libre llamada Weka, dedicada a la minería de datos. En segundo lugar, integrar el algoritmo CHAID* propuesto recientemente por el grupo ALDAPA como una adaptación del anterior.

Para probar la validez de ambas implementaciones, el proyecto contará con una fase de experimentación donde se expondrán a una extensa colección de problemas de clasificación y se analizarán los resultados obtenidos.

ÍNDICE DE CONTENIDO

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1. INTRODUCCIÓN

1.1. Motivación del proyecto

Una de las características de los sistemas de información actuales es la ingente cantidad de datos que son capaces de generar y almacenar. Esto no solo se debe a los grandes avances tecnológicos que han disparado la capacidad de cómputo y almacenamiento de los ordenadores, sino también a la expansión de la informática a prácticamente todos los ámbitos de la vida: el ámbito doméstico y social, el ámbito industrial, el de las comunicaciones, el financiero, el médico, etc.

El valor de esta gran cantidad de datos está limitado por la capacidad que tengamos para procesarlos, interpretarlos y extraer información que resulte útil para comprender hechos pasados, predecir hechos futuros y ayudar en la toma de decisiones.

Esto ha provocado que muchos esfuerzos en el mundo de la Informática se hayan centrado en el tratamiento de cantidades masivas de datos con objetivos como la generación de informes estadísticos, el reconocimiento de patrones o la generación de modelos explicativos y predictivos con aplicación en multitud de campos: finanzas, medicina, meteorología, sociología...

De esta manera surge la *Minería de datos* -o *Data Mining*-, que consiste en un campo multidisciplinar que combina el mundo de la Estadística Clásica y sus técnicas para el reconocimiento de patrones, con el mundo de la Inteligencia Artificial y sus técnicas de aprendizaje automático.

Los objetivos principales de la Minería de Datos pueden resumirse en [Per-99]:

- Particionado y/o agrupación de conceptos/datos similares entre sí.
- Generación de modelos explicativos o predictivos de una variable en base a otras.
- Búsqueda de patrones y subpatrones frecuentes.
- Búsqueda de tendencias y desviaciones.
- Búsqueda de relaciones entre variables.

En el presente proyecto se profundizará, sobre todo, en un paradigma concreto de la Minería de datos: el aprendizaje automático basado en árboles de decisión. Dentro de este paradigma se profundizará en un algoritmo en particular llamado CHAID, así como en una extensión propuesta por el grupo ALDAPA bajo el nombre CHAID*.

También se presentará una de las aplicaciones de software más conocidas para *Data Mining* y *Machine Learning*, Weka, donde finalmente se implementarán las dos versiones mencionadas del algoritmo CHAID: la versión clasica y la propuesta por ALDAPA.

1.2. Organización de la memoria

Tras esta introducción, el Capítulo 2 describe los antecedentes de los que parte este proyecto y los objetivos que se propone.

El capítulo 3 realiza una introducción teórica a las áreas del Análisis de Datos y el Aprendizaje Automático.

El capítulo 4 profundiza en el paradigma de los árboles de clasificación y explica con más detalle tres algoritmos concretos: C4.5, CHAID y CHAID*.

El capítulo 5 explica la plataforma Weka y las opciones de desarrollo que ofrece.

El capítulo 6 expone la parte más práctica del proyecto, explicando la implementación del algoritmo CHAID realizada en Weka, así como su adaptación CHAID*.

En el capítulo 7 se ofrecen los resultados de la experimentación realizada con el algoritmo CHAID*.

Finalmente, los capítulos 8 y 9 cierran esta memoria con las conclusiones del proyecto y la bibliografía referenciada, respectivamente.

2. DOCUMENTO DE OBJETIVOS DEL PROYECTO (DOP)

2.1. Antecedentes

El grupo ALDAPA (ALgorithms, DAta mining and PArallelism) es un grupo de investigación de la UPV/EHU que centra su trabajo en el área de la minería de datos, entre otras.

Actualmente el grupo cuenta con varias lineas estables de investigación, entre las que se encuentra la de “Algoritmos de aprendizaje supervisado con capacidades explicativas”. En esta línea el grupo diseña y trabaja con algoritmos de aprendizaje que pueden afrontar problemas que requieren de una explicación de la clasificación realizada, además de una eficiente clasificación [Mug-16].

Uno de estos algoritmos es CHAID, del que recientemente han propuesto una extensión en [Iba-15] bajo el nombre de CHAID*. Como se comenta en dicha publicación, a pesar de ser un algoritmo clásico, desde el inicio solo se distribuyó dentro de aplicaciones de software comerciales como SPSS o KnowledgeSeeker. Aunque actualmente también se puede encontrar en algunos proyectos de software libre como R, sigue ausente en otras plataformas ampliamente extendidas como Weka.

El grupo ALDAPA también investiga en contextos en los que la cantidad de casos de alguna de las clases está muy desbalanceada con respecto al resto, lo que es un inconveniente para la mayoría de los paradigmas de aprendizaje automático [Mug-16].

En esta línea, en [Arb-13] publicaron el algoritmo CTC, donde se extiende otro algoritmo clásico llamado C4.5 aplicando el concepto de “consolidación”, e implementándolo en la plataforma Weka bajo el nombre J48Consolidated. La motivación de este trabajo fue hacer “pública y disponible” la implementación de CTC.

También se ha publicado una versión “consolidada” del ya mencionado CHAID bajo el nombre CTCHAID en [Iba-15b], pero al igual que ocurre con CHAID*, hasta ahora no se han integrado en Weka: como se ha dicho, el algoritmo original no se incluye actualmente en dicho software, así que no existe una base que se pueda tomar como punto de partida, como sí ocurría con C4.5.

Por tanto, en este proyecto se plantea llevar a cabo la implementación de CHAID en la plataforma Weka como base para implementar otras versiones planteadas por ALDAPA como CHAID* y CTCHAID.

2.2. Objetivos

Con estos antecedentes y buscando definir un alcance asumible en el tiempo disponible para la realización de este proyecto, se establecen los siguientes objetivos:

- Aprender los conceptos teóricos fundamentales para entender los algoritmos de clasificación en general y el CHAID en particular.

- Estudiar la arquitectura de Weka y la manera en que se pueden integrar en ella nuevos algoritmos de clasificación.

- Realizar en Weka una implementación del algoritmo CHAID clásico.

- Extender el clasificador desarrollado con una implementación del algoritmo CHAID*.

- Demostrar la validez de la implementación de ambos clasificadores mediante las herramientas de experimentación de Weka.

3. CONCEPTOS TEÓRICOS

En este capítulo se introducirán algunos conceptos teóricos necesarios para la comprensión de los algoritmos CHAID y CHAID* implementados en este proyecto. También se dará una visión global del Aprendizaje Automático y sus distintas ramas para poner en contexto el tipo de técnicas en las que se enmarcan dichos algoritmos.

3.1. Conjuntos de datos

3.1.1. Conceptos básicos

El primer concepto a introducir es el de *conjunto de datos* o *población*, que hace referencia al total de la información disponible acerca del problema a resolver. Estos conjuntos de datos se componen de una serie de *casos* o *individuos* concretos.

A su vez, estos individuos se describen mediante una serie de características llamadas *variables* o *atributos*, que en cada caso tomarán unos valores en particular.

Un ejemplo podría ser un listado de alumnos de la facultad que recogiese su edad, sexo y las notas obtenidas en una serie de asignaturas. El listado sería el *conjunto de datos*, cada alumno sería un *caso* o *individuo*, y la edad, sexo y asignaturas serían los *atributos* o *variables*.

Las variables se pueden dividir en dos tipos: independientes o dependientes. Las primeras, también llamadas *predictoras* o *explicativas*, son las variables que realmente describen al individuo, e influyen en otras variables. Las

segundas, también llamadas variables *clase* -término que se usará de aquí en adelante para referirse a ellas- son precisamente las predichas o explicadas por las variables independientes. A las categorías o valores que puede adoptar la variable independiente de un conjunto de datos también se les conoce como las *clases* del problema.

En el contexto de los problemas de clasificación, lo que se pretende es predecir la clase de un individuo dado a partir de los valores de sus atributos independientes. Para ello deberá buscar una correlación entre variables lo suficientemente significativa desde un punto de vista estadístico como para poder asumir, con cierto nivel de seguridad, que los valores del individuo para sus variables independientes implican su pertenencia a una clase particular.

Siguiendo con el ejemplo anterior, si se encuentra una relación estadísticamente significativa entre las notas de ciertas asignaturas de primer año -variables independientes-, con los aprobados y suspensos de una asignatura de segundo año -variable dependiente- podría predecirse, a partir de las notas obtenidas por un alumno cualquiera en su primer año, si éste aprobará o suspenderá en el siguiente curso esa asignatura analizada.

3.1.2. Tipos de variables

Además de por el rol que desempeñan en la clasificación -dependiente o independiente-, las variables se pueden clasificar según el tipo de datos que almacenan.

Por un lado están las variables nominales, es decir, variables cuyos posibles valores son un conjunto de categorías con nombre. En el ejemplo anterior, el atributo *sexo* de los alumnos sería una variable nominal donde sus posibles valores serían *hombre* o *mujer*. Por tanto, son variables descriptivas, que no

expresan una característica cuantificable, ni tampoco existe ningún orden ni relación entre los valores de este tipo de atributos.

Por otro lado están las variables *ordinales* donde, al contrario que en las *nominales*, sí existe un orden entre sus posibles valores. Un ejemplo sería el atributo *nivel de estudios*, donde podrían definirse como posibles valores *primarios*, *secundarios* y *universitarios*. En este caso sí existiría una relación de orden -los estudios secundarios serían superiores a los primarios, pero inferiores a los universitarios-.

Además, los valores ordinales pueden ser numéricos, siempre que se muevan en rangos de valores "manejables". Por ejemplo, en un atributo *nota obtenida en X asignatura* que tomara como posibles valores un número entero del 0 al 10. El número de posibilidades está bastante acotado y también existe una relación de orden, ya que obviamente, tener una nota de 10 es mejor que tener un 0, etc.

Los dos tipos de variables mencionados hasta ahora -*nominales* y *ordinales*- también se conocen en algunos contextos como *variables discretas*. Cabe mencionar que la mayoría de algoritmos de clasificación no hacen distinción entre variables *nominales* y *ordinales*, sino que tratan ambos casos como *nominales*. Sin embargo, el algoritmo CHAID sí hace esta distinción y por lo tanto es importante tenerla en cuenta para este proyecto.

Sin embargo, cuando el atributo almacena valores numéricos que se mueven en un rango lo bastante elevado -incluso, infinito-, se consideran variables *continuas*. Este tipo de variables siempre expresan características cuantificables y con relación de orden.

3.1.3. Valores *missing*

Además de los posibles valores que para cada individuo puede tomar un atributo según su tipo -*nominal*, *ordinal* o *contínua*-, existe otra posibilidad: que no tome ningún valor en absoluto.

Este es un caso especial al que puede llamarse *caso perdido* o *missing value*, como se le llamará aquí a partir de ahora. Expresa la ausencia de valor alguno para el atributo en un determinado caso. Como se verá más adelante, hay algoritmos que hacen un tratamiento de este tipo de casos y otros, en cambio, necesitan que todos los casos tengan valores conocidos.

3.2. Aprendizaje automático

El aprendizaje automático es una rama de la Inteligencia Artificial donde se implementan algoritmos que procesan conjuntos de datos, identifican patrones y extraen conocimientos suficientemente genéricos como para aplicarlos a nuevos conjuntos de datos.

Dicho de otra manera: se trata de algoritmos que hacen que las máquinas “aprendan”.

Los algoritmos de aprendizaje automático se pueden dividir en dos grandes bloques: por un lado el *Aprendizaje No Supervisado* o *Análisis Clúster*, y por otro lado el *Aprendizaje Supervisado* o *Reconocimiento de patrones*.

3.2.1. Aprendizaje Supervisado

Este tipo de aprendizaje es el que se realiza partiendo de un conjunto de datos en el que ya se conoce la variable clase de los individuos. A esta población inicial se le llamará *conjunto de entrenamiento*.

Los algoritmos que pertenecen a este bloque intentan encontrar relaciones entre las variables independientes y las clases del problema. Con las relaciones o patrones que logren encontrar en el conjunto de entrenamiento, construirán un modelo de clasificación lo suficientemente genérico como para ser capaz de clasificar correctamente nuevos casos en los que ya no se conozca el valor de la variable clase.

3.2.2. Aprendizaje No Supervisado

En este tipo de aprendizaje, en cambio, no existe un conocimiento previo a partir del cuál se pueda “entrenar” a la máquina para resolver nuevos problemas, es decir, se desconoce el valor de la variable dependiente en el conjunto de datos.

Los algoritmos de este tipo se encargan de buscar agrupaciones de casos en el conjunto de entrenamiento en base al valor de sus variables independientes, buscando que los casos del mismo grupo se parezcan y que los casos de distinto grupo sean lo más diferentes posible. Podría decirse entonces que las *agrupaciones* o clústers identificados por el algoritmo representarían a las clases del conjunto de datos.

Un hecho característico es que, en este tipo de aprendizaje, a estas clases no se les asigna ningún valor en particular. El algoritmo solamente identifica la existencia de estas agrupaciones, y es el investigador quien puede darles una interpretación o asignarles valores concretos.

Por ejemplo, la Figura 3 1 ilustra una clasificación no supervisada. En él se toma una muestra de individuos y se analiza la relación entre su altura y peso. El resultado son dos grupos o *clústers* claramente diferenciados, aunque el modelo no da un valor semántico a estas agrupaciones; simplemente revela su existencia.

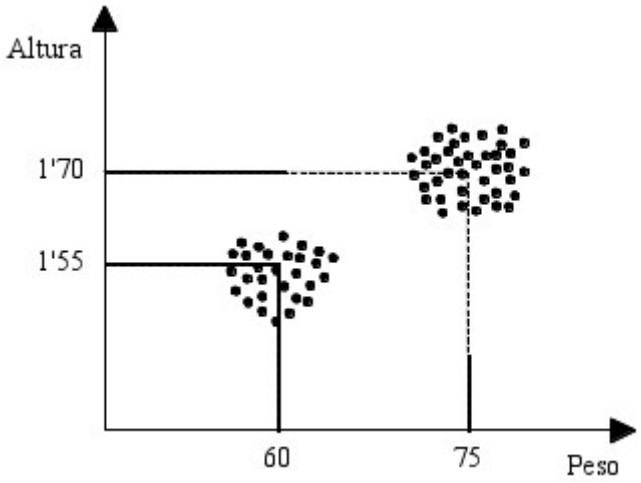

Figura 3 1: Ejemplo de aprendizaje no supervisado

Ahora bien, el investigador que analice estos resultados puede concluir que lo que representan estos *clústers* es la diferencia de altura y peso entre hombres y mujeres, por ejemplo. Estas conclusiones podrían dar pie a la posterior elaboración de un análisis supervisado, donde esta muestra sirviera de conjunto de entrenamiento y, ahora sí, los casos tengan una variable de clase -en este caso, *sexo* del individuo- con unos posibles valores -*hombre* o *mujer*-, y a partir de ahí se le podrían dar nuevos casos al modelo para los que se conozcan los valores de los atributos independientes -*altura* y *peso*- para que los clasifique en una de esas dos categorías.

Sirva este ejemplo como explicación genérica del tipo de Aprendizaje No Supervisado, pero queda fuera de los objetivos de este proyecto el profundizar más en este tipo de modelos. En adelante nos centraremos en el aprendizaje supervisado.

3.2.3. Modelos de aprendizaje supervisado

Dentro del bloque del aprendizaje supervisado existen varios modelos de clasificación. De nuevo, pueden distinguirse dos tipos de modelos:

Por un lado están los *Modelos No Explicativos*, entre los que se encuentran paradigmas de clasificación como el análisis discriminante, las redes neuronales, técnicas basadas en medidas de distancia (k-NN), etc. Estos son modelos que resuelven el problema de clasificación sin aportar información añadida, como podría ser qué características son las que han hecho que a un patrón o caso dado se le haya asignado una determinada clase.

Por otro lado están los *Modelos Explicativos*, que además de realizar la clasificación de un problema aportan una justificación del resultado obtenido. Ejemplos de este tipo de modelos son las reglas de inducción y los árboles de clasificación.

Este proyecto se centra en el paradigma de los árboles de clasificación. Éste es un modelo de clasificación para el que resulta sencillo ver en qué sentido el modelo es explicativo; basta con representar visualmente un árbol de decisión como el de la Figura 3 2. En cada nodo, partiendo del nodo raíz, se pregunta por el valor de un atributo concreto del caso a clasificar, este valor determinará cuál de las ramas del nodo escoger, y así sucesivamente hasta llegar a un nodo hijo, que siempre corresponde a una clase concreta. Por tanto es un modelo que *explica* la pertenencia del caso a una clase basándose en que ciertos atributos del caso tienen ciertos valores.

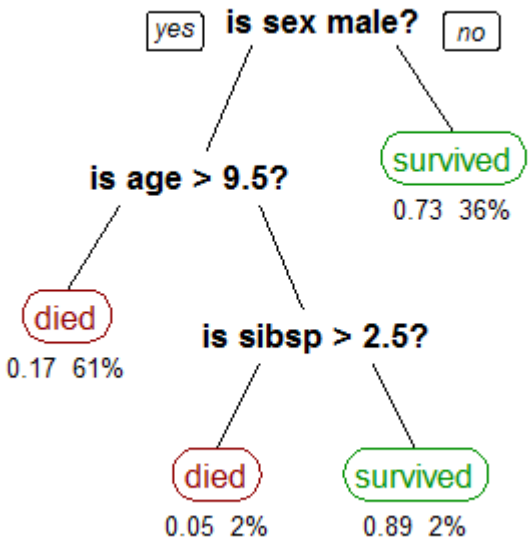

Figura 3 2: Ejemplo de árbol de clasificación (fuente: Wikipedia).

3.3. Técnicas de validación

El valor de un clasificador reside fundamentalmente en su capacidad de generalización, de modo que sea capaz de identificar en la muestra de entrenamiento patrones que después le sirvan para clasificar correctamente otros casos de clase desconocida. A esto se le llama *bondad del clasificador*, y es lo que intenta cuantificarse mediante las técnicas de validación.

La técnica más básica podría consistir en ejecutar el clasificador contra la base de datos de entrenamiento y contabilizar el número de errores que comete, aprovechando que es un conjunto de datos para el que conocemos la clase de todos los casos. Sin embargo esta tasa de error, conocida como *Error de entrenamiento*, no es un criterio de bondad demasiado valioso porque se presupone que será una tasa de error más bien optimista, al estar clasificando los mismos casos con los que se ha entrenado el modelo.

Una tasa de error mucho más interesante de calcular es el *Error poblacional*, que expresa el error cometido al clasificar una población universal de individuos desconocidos para el modelo. A continuación se expondrán algunas de las técnicas más comunes para calcular este criterio de bondad.

3.3.1. Método H (Holdout)

Este método consiste en dividir la base de datos en dos conjuntos. El primer conjunto, llamado *conjunto de entrenamiento* y que suele comprender 2/3 de la base de datos, se utilizará para construir el modelo. El segundo conjunto, llamado *conjunto de test* y que comprende el 1/3 de datos restante, será sobre el que se aplique el modelo construido para calcular su bondad.

Es un método sencillo y no resulta tan robusto como otras técnicas, sin embargo sí que es mucho más eficiente. Métodos más avanzados como los que se explicarán a continuación resultan mucho más caros desde un punto de vista computacional, sobre todo a medida que aumentan el número de casos y/o atributos, hasta el punto de que para bases de datos realmente grandes el Holdout puede ser el único método de validación viable.

3.3.2. Validaciones cruzadas

Se trata de un tipo de técnicas donde nunca se testea el clasificador con individuos que hayan formado parte de la muestra de entrenamiento. A continuación se explicarán las dos técnicas más conocidas de este tipo.

3.3.2.1. Leave-one-out

Esta técnica se basa en la idea de las *validaciones cruzadas.* Partiendo de una base de datos de tamaño “n”, ésta se dividirá en “n” submuestras. Para cada submuestra se separa un único individuo, y con el resto de individuos se construye un modelo con el que después se intentará clasificar el individuo separado. Al no haber sido parte del conjunto de entrenamiento de la submuestra, si se clasifica mal este individuo se puede considerar un *error poblacional*.

Finalmente, se construirá un clasificador con la base de datos completa y, según esta técnica de validación, su error poblacional será la media de los errores poblacionales cometidos para las “n” submuestras con las que se ha realizado el test.

3.3.2.2. m-fold cross-validation

Esta técnica parte de la misma idea que el *Leave-one-out*, pero en lugar de separar un único individuo de cada submuestra separará un número “n” de individuos. Este grupo de “n” individuos se denomina *fold* y, de la misma manera que en el anterior método, se usarán para medir el error poblacional de la submuestra a la que pertenecen.

De la misma manera, el clasificador final se construye con la base de datos completa y se calcula su error poblacional como la media de los errores poblacionales de las submuestras utilizadas.

4. ÁRBOLES DE CLASIFICACIÓN

4.1. Introducción

Tal como se introdujo en el capítulo 3, los árboles de clasificación son un paradigma del Aprendizaje Automático que se enmarca dentro de los algoritmos de *Aprendizaje Supervisado* y que sirve para generar modelos de clasificación *explicativos*.

El objetivo de estos algoritmos es el de, partiendo de un conjunto de entrenamiento, generar una estructura de *árbol de decisión* que hará de *modelo de clasificación*. Una vez construído el árbol, el proceso de clasificación es tan simple como dar al modelo, como datos de entrada, los valores de los atributos para un nuevo caso. En cada nodo del árbol se comprobará el valor de un atributo concreto, decidiendo así por qué rama avanzar. Finalmente se llegará a un nodo hoja, que siempre representa un valor de la variable clase.

Dos de los algoritmos más extendidos son el C4.5 y el CHAID. En este capítulo se explicarán ambos en mayor profundidad, y además se introducirá el algoritmo CHAID*, la adaptación del CHAID propuesta por el grupo ALDAPA.

Para explicar los tres algoritmos de una forma más comparativa, para cada uno de ellos se analizarán cuatro características clave:

- **Función de división o *split function***: Criterio seguido para dividir cada nodo del árbol en varios subnodos. Hay que señalar que, hasta cierto punto, un mayor número de subnodos harán que el árbol sea más preciso, pero también que sea más complejo y menos generalista, con lo que es importante que los algoritmos tengan un buen criterio a la hora

de decidir si dividen un nodo, y en base a qué atributos y valores lo hacen.

- **Tipo de variables**: se indicará con qué tipo de datos puede trabajar cada algoritmo: discretas -nominales u ordinales- y/o continuas.

- **Valores *missing*:** qué hace cada algoritmo ante casos del conjunto de datos con valores ausentes.

- **Poda del árbol**: técnicas empleadas para reducir la complejidad del árbol generado, es decir, el número de nodos en que se divide.

4.2. ALGORITMO C4.5

Se trata de un algoritmo desarrollado por Ross Quinlan en 1993 [Qui-93] como extensión del ID3, desarrollado en 1986 por el mismo autor [Qui-86]. Ambos se basan en el concepto de *entropía* a la hora de construir el árbol.

Sus principales características son [Iba-15]:

- **Función de división**: Usa el concepto de entropía como criterio discriminante para determinar cuál es el mejor atributo para dividir el nodo actual. A menor *entropía* de un atributo, mayor *ganancia de información* aporta y por tanto más adecuado es utilizarlo para hacer la división.

- **Tipos de variables**: Puede tratar variables discretas y continuas.
    - Para las variables discretas, genera un nodo hijo por cada posible valor del atributo.
    - Para las variables continuas, se busca el mejor punto de corte basándose en la *ganancia de información* -concepto relacionado con la entropía-, con el cual genera una división binaria.

- **Valores *missing***: el algoritmo asigna un peso de 1 en la raíz del árbol a cada ejemplo del conjunto de entrenamiento. Cuando en un nodo aparece un caso con valor *missing* para el atributo elegido para hacer la división, este caso se asigna a todos los nodos hijos y se le da, en cada uno, un peso proporcional al número de casos con valores conocidos que contiene ese nodo hijo.

- **Poda del árbol**: para reducir la complejidad del árbol construído, C4.5 aplica una estrategia de poda llamada *reduced error prunning* que simplifica el modelo.

4.3. ALGORITMO CHAID

Como se ha dicho, este es el algoritmo cuya implementación en WEKA supone uno de los objetivos de este proyecto. Los detalles de dicha implementación, así como la propia plataforma Weka, se describirán en los capítulos 5 y 6.

CHAID consiste en un algoritmo para la construcción de árboles de decisión basado en el testeo de significancia ajustada. Su nombre proviene de “Chi-squared Automatic Interaction Detection”. Fue publicado en 1980 por Gordon V. Kass y proviene del clásico AID, con algunas particularidades añadidas [Kas-80].

Se distingue por el uso de la chi-cuadrado, χ^2, para medir el grado de correlación entre las variables independientes y la clase.

Estas son sus principales características:

- **Función de división**: Usa la prueba χ^2 de Pearson para decidir si la diferencia entre una variable predictora y la clase es suficientemente significativa, y si por tanto merece la pena dividir el nodo en base a dicha variable.

- **Tipos de variables**: Solo puede tratar variables discretas. Sin embargo, está diseñado para diferenciar entre variables *nominales* -donde, como se explicaba en el apartado 3.1.2, los valores no siguen un orden determinado- y *ordinales* -donde sí lo siguen-. Utiliza un heurístico para encontrar la mejor combinación de categorías para dividir el nodo. La diferencia en el tratamiento de unas y otras reside en que el algoritmo de Unión/División de categorías de Kass puede realizar cualquier

combinación posible de valores nominales, mientras que en el caso de los ordinales solo agrupa valores consecutivos. En el siguiente apartado se explica este algoritmo con más detalle.

- **Valores *missing***: CHAID considera los *missings* como uno más entre los valores que puede tomar un atributo. Para cada nodo, en un primer momento los *missing* se dejan aparte, mientras se está buscando la mejor combinación de valores para realizar la división. Una vez encontrada la mejor combinación sin *missings*, se buscará combinarlos con el grupo cuya diferencia sea menos significativa según el criterio de la χ^2 -es decir, el grupo al que más se parezcan-, aunque si no existe ninguno con un nivel de significancia adecuado se dejarán en un grupo aparte.

- **Poda del árbol**: el caso de CHAID, no se aplica una posterior poda del árbol como ocurría en C4.5, aunque tiene varios mecanismos para autorregular el crecimiento del árbol generado que podrían considerarse una *pre-proda*: Como se ha dicho, la prueba χ^2 de Pearson sirve como criterio para dividir o no un determinado nodo. Además, el algoritmo tiene algunos parámetros que determinan la cantidad mínima de casos que debe tener un nodo para poder ser dividido, o la cantidad mínima de casos que debe tener cada uno de los nodos hijo.

4.3.1. Algoritmo de Unión / División de categorías de Kass

Puesto que implementar el algoritmo CHAID supone uno de los objetivos principales de este proyecto, merece la pena ofrecer una descripción más detallada del mismo.

Este algoritmo tiene tres fases: unión, división y parada [Per-99]. Su principal diferencia respecto a otros algoritmos de clasificación como el C4.5 reside en el algoritmo de “Unión / División de categorías” que aplica dentro de la primera fase para cada uno de los nodos generados, partiendo del nodo raíz.

A continuación se explican las tres fases del algoritmo:

Fase 1: Unión (Merging)

Para cada variable independiente se realizan los siguientes pasos:

1. Generar la tabla de contingencia de todas las categorías de la variable en curso contra las de la variable dependiente.

2. Para cada par de categorías -o cada par de “grupos de categorías”, en posteriores pasos por este punto- que se pueda combinar, calcular el estadístico χ^2 para el test de independencia, *pairwise chi-square*.

3. Calcular el valor p de probabilidad para cada ‘pairwise chi-square‘. Si este valor no es suficientemente significativo para algún par, es decir, si es mayor que un cierto umbral de unión de categorías, se escoge el par de mayor valor de probabilidad (con lo que su χ^2 será el más pequeño) y se unen ambas categorías (o grupos) en un mismo grupo, continuando en el paso 4. Si todos los pares son significativos, saltar al paso 5.

4. Para todo nuevo grupo formado por la unión de 3 o más de las categorías originales, realizar todas las posibles combinaciones de división binaria de este grupo. Calcular el estadístico 2 para cada posibilidad y si alguna es suficientemente significativa (según el mismo valor umbral tomado en el punto 3), dividir el grupo y volver al paso 2. En caso de que hubiera más de una división significativa, se elegiría la de la

χ^2 más grande (con valor de probabilidad; el menor). Si no hubiese ninguna, se continuará por el siguiente paso, paso 5.

5. Cualquier categoría (o grupo) con menor número de observaciones (individuos o casos), que el valor mínimo establecido se deberá unir con aquel grupo que sea más parecido a él, medido esto por el que tenga el 'pairwise chi-square' más pequeño.

    5.a. Si la muestra contiene *missing values* para la variable en curso, también se intentarán unir con aquel grupo que sea más parecido al suyo. En este caso, además de buscar el grupo con el 'pairwise chi-square' sea más pequeño, se comprueba que la unión sea estadísticamente significativa. En caso de que ninguna de las posibles uniones lo sea, se dejan los *missing values* como un grupo aparte.

6. Una vez obtenido la combinación de grupos de categorías definitiva, realizaremos el test para ésta, obteniendo el valor de χ^2 y su valor de probabilidad asociado. A éste último se le suele ponderar (opcional en las herramientas que implementan este algoritmo) por el coeficiente de Bonferroni, dándose en llamar entonces, valor de probabilidad ajustado.

   Este coeficiente de Bonferroni es un corrector asociado al valor de χ^2 de la tabla de contingencia resultante, que hace al algoritmo más precavido en las decisiones que toma en cuanto a la significancia de la relación entre las variables, y se aplica mediante una de tres ecuaciones propuestas en [Kas-80], dependiendo del tipo de variable con la que se esté trabajando.

Fase 2: División (Splitting)

Elegir entre los posibles predictores estadísticamente significativos, aquél cuyo valor de probabilidad ajustado sea el menor. En función de cómo hayan quedado combinados los grupos de categorías para éste, se dividirá la subpoblación del nodo del árbol en análisis en distintos estratos, correspondiendo éstos con cada uno de los grupos. Si no existiera ninguna variable que fuera suficientemente significativa, no habría división, con lo que el nodo en curso quedaría como nodo hoja del árbol.

Fase 3: Criterio de Parada (Stopping)

Si el nodo en estudio ha sido estratificado, se saltaría al primer nodo hijo de éste comenzando de nuevo por la Fase 1 de este algoritmo. Si no ha sido así, el siguiente nodo en el análisis será el que corresponda recorriendo el árbol en pre-orden. En ambos casos, el siguiente nodo ha de tener un número mínimo de observaciones para poder tomar parte en el análisis, si no, constituirá otro nodo hoja del árbol.

En [Kas-80] se asegura que el procedimiento converge a una solución estable en un número finito de pasos del algoritmo. En la práctica, una unión raramente se divide, pero se debe permitir para asegurar resultados cercanos al óptimo [Per-99]. En algunas implementaciones de este algoritmo, el paso 4 de la fase de Unión -donde se vuelve a plantear dividir en dos un grupo de categorías- es un paso opcional [Ibm-12], lo cual posibilita una reducción considerable de tiempo cuando el número de categorías del atributo es muy elevado. Por este motivo y por la complejidad que añade, se ha decidido dejar este paso fuera del alcance del proyecto.

4.4. ALGORITMO CHAID*

Este algoritmo es una adaptación del CHAID pensado para dotar al original de algunas capacidades de las que carecía respecto a otros algoritmos, como C4.5. Fue propuesta recientemente por el grupo ALDAPA en [Iba-15] con la idea de poder usarlo con las mismas bases de datos que esos otros algoritmos.

Sus principales características son:

- **Función de división**: mantiene el criterio del CHAID original, es decir χ^2 como criterio de división.

- **Tipos de variables**: puede tratar con variables discretas -nominales y ordinales- y continuas. Para éstas últimas usa la misma estrategia que el C4.5, con la diferencia de que para éstas también usa la χ^2 como criterio para buscar el punto de corte, en lugar de la *ganancia de información* del C4.5.

- **Valores *missing***: en el caso de las variables continuas, se comporta igual que para las discretas: evaluará la posibilidad de unirlos a uno de los grupos en que se divida el nodo -en este caso siempre serán dos: los valores anteriores al punto de corte, y los valores posteriores-, pero si ninguna de estas uniones resulta suficientemente significativa, se dejan en un grupo aparte.

- **Poda del árbol**: en este caso se introduce la misma técnica de post-poda que en el C4.5 -*reduced error prunning*-, ya que, a pesar de que el CHAID original se limita a las técnicas de *pre-poda* explicadas en el apartado anterior, los experimentos con esta versión del algoritmo

mostraron árboles bastante complejos con peores resultados de clasificación para variables contínuas.

5. WEKA

5.1. Introducción

Weka es, según la definición dada en su propia página web, "*una colección de algoritmos de aprendizaje automático para tareas de minería de datos*" [Hal-09a]. La figura 5 1 muestra el logotipo de la aplicación.

Estos algoritmos pueden ejecutarse directamente desde el cliente que ofrece el propio software -a través de alguna de las interfaces que explicaremos en el apartado 5.3- o bien llamándolos desde el código de otras aplicaciones Java, a modo de librería externa.

Figura 5 1: Logotipo de Weka.

Esta plataforma contiene herramientas para el preprocesado de conjuntos de datos, clasificación, regresión, agrupamiento o *clustering*, reglas de asociación y visualización de datos. Las fuentes de datos con las que pueden ejecutarse estas herramientas pueden provenir de ficheros en diferentes formatos -en el apartado 5.4 veremos el más comúnmente utilizado, el formato .ARFF-, así como de bases de datos relacionales a través de una interfaz JDBC.

Aparte de los usos que se le pueden dar desde el punto de vista de usuario, la característica que hace a Weka una plataforma interesante para el presente proyecto es su carácter abierto: se trata de un software libre distribuído bajo la licencia "*GNU General Public License*", que da libertad para utilizar el software, modificarlo, y redistribuirlo con tus modificaciones [Smi-07].

Respecto al punto de modificar el software, el propio diseño de la plataforma la hace fácilmente extensible permitiendo implementar nuevos algoritmos, filtros, etc., siguiendo su arquitectura orientada a objetos, como se explicará en mayor detalle en el capítulo 6.

5.2. Motivación

Se ha mencionado la flexibilidad que ofrece Weka para extender su código mediante la integración de, por ejemplo, nuevos clasificadores en la colección que de por sí ofrece. En el presente apartado se razonará por qué, más allá de estas facilidades, resulta interesante para un proyecto como éste programar un algoritmo como parte de un software ajeno en lugar de desarrollar uno propio.

Es cierto que el desarrollo de un software propio ofrece una serie de libertades, como la de poder elegir la tecnología usada, el lenguaje de programación, diseñar la arquitectura… Sin embargo, hay varias razones por las que resulta interesante la implementación de CHAID* dentro de la plataforma Weka:

Difusión: Al tratarse de un software bastante popularizado, especialmente en el ámbito educativo, resulta más fácil distribuir nuevos paquetes de Weka y permitir que otras personas los utilicen como parte de este software, obteniendo mayor visibilidad que si se distribuyera como un software aparte.

Interfaz: Como se ha mencionado en la introducción a este capítulo, Weka cuenta con una interfaz gráfica desde la que ejecutar los clasificadores así como una amplia colección de herramientas. Simplemente con asegurar que tus paquetes implementan una serie de interfaces ofrecidas por Weka -sobre las que se entrará en detalle en el apartado 6.1-, el paquete queda integrado en la plataforma de manera que puede usarse la interfaz gráfica para ejecutarlo, cambiar sus opciones, examinar los resultados con las herramientas de visualización de Weka, etc. Es decir, que el clasificador queda enriquecido por una serie de utilidades que de otra manera serían muy costosas de implementar.

Benchmarking: Otro valor añadido consiste en la posibilidad de incluir el nuevo clasificador en bancos de pruebas donde, por ejemplo, se compare su rendimiento frente a distintas bases de datos, o incluso se comparen los resultados que ofrece frente a otros clasificadores de la plataforma, como se mostrará en el apartado 5.3.2.

Un ejemplo de esto es el capítulo 7 de esta memoria, dedicado al testeo exhaustivo de los algoritmos implementados, entre cuyos resultados se ha podido incluir información y comparativas gracias, precisamente, a haberlo realizado mediante el banco de pruebas de Weka.

5.3. Interfaz de usuario

Como se ha mencionado en la introducción a este capítulo, la plataforma Weka ofrece diversas herramientas que pueden utilizarse, entre otras maneras, a través de una interfaz gráfica. En este apartado se explicará el modo de empleo de algunas de estas herramientas y sus correspondientes opciones.

El objetivo no es hacer un repaso exhaustivo de todas opciones que ofrece Weka, sino ofrecer una breve guía de aquellas que se han utilizado durante el desarrollo de este proyecto para las pruebas y depuración del código. De este modo, se dará a conocer la manera más sencilla de ejecutar los algoritmos implementados -así como el resto de algoritmos que ya formaban parte del código base de Weka- pudiendo así probarlos con diferentes conjuntos de datos, compararlos con otros clasificadores, verificar los resultados de la experimentación ofrecidos en el capítulo 7, etc.

Figura 5 2: Pantalla inicial de Weka.

Lo primero que se verá al abrir el programa es una pequeña ventana (Figura 5 2) desde la que se puede acceder a las cuatro aplicaciones en las que se

divide: *Explorer*, *Experimenter*, *Knowledge Flow* y *Simple Cli*. De estas cuatro aplicaciones, las dos primeras han sido las únicas utilizadas durante el desarrollo de este proyecto y, por lo tanto, las que abordaremos en este apartado.

Por un lado, el *Explorer* es la aplicación más interesante de cara a probar y depurar el código durante su fase de implementación, ya que permite lanzar ejecuciones individuales de un algoritmo concreto sobre una base de datos en particular, tras lo que imprime en pantalla abundante información sobre los resultados de la ejecución.

Por otro lado, el *Experimenter* es una aplicación de gran utilidad para poner a prueba los algoritmos de clasificación una vez terminados de implementarse, ya que permite preparar bancos de pruebas donde se podrán visualizar y comparar los resultados para múltiples conjuntos de datos o, incluso, realizar comparaciones con otros algoritmos.

5.3.1. Explorer

Al abrir la aplicación Explorer se accede a una ventana con seis pestañas: *Preprocess*, *Classify*, *Cluster*, *Associate*, *Select attributes* y *Visualize* (Figura 5 3).

5.3.1.1. Pestaña *Preprocess*

La pestaña *Proprocess* permite seleccionar el conjunto de datos con el que se va a trabajar y ofrece algunas opciones para la visualización y manipulación de los mismos.

El primer paso aquí será pulsar en las opciones "*Open file*", "*Open URL*", "*Open DB*" o "*Generate*" dependiendo de la fuente de datos que se quiera

utilizar. Por ejemplo, en este proyecto se ha trabajado con bases de datos .ARFF descargadas desde la página web de Weka [Hal-09b], por lo que en este caso se utilizaría la primera opción para seleccionar el fichero de datos a cargar.

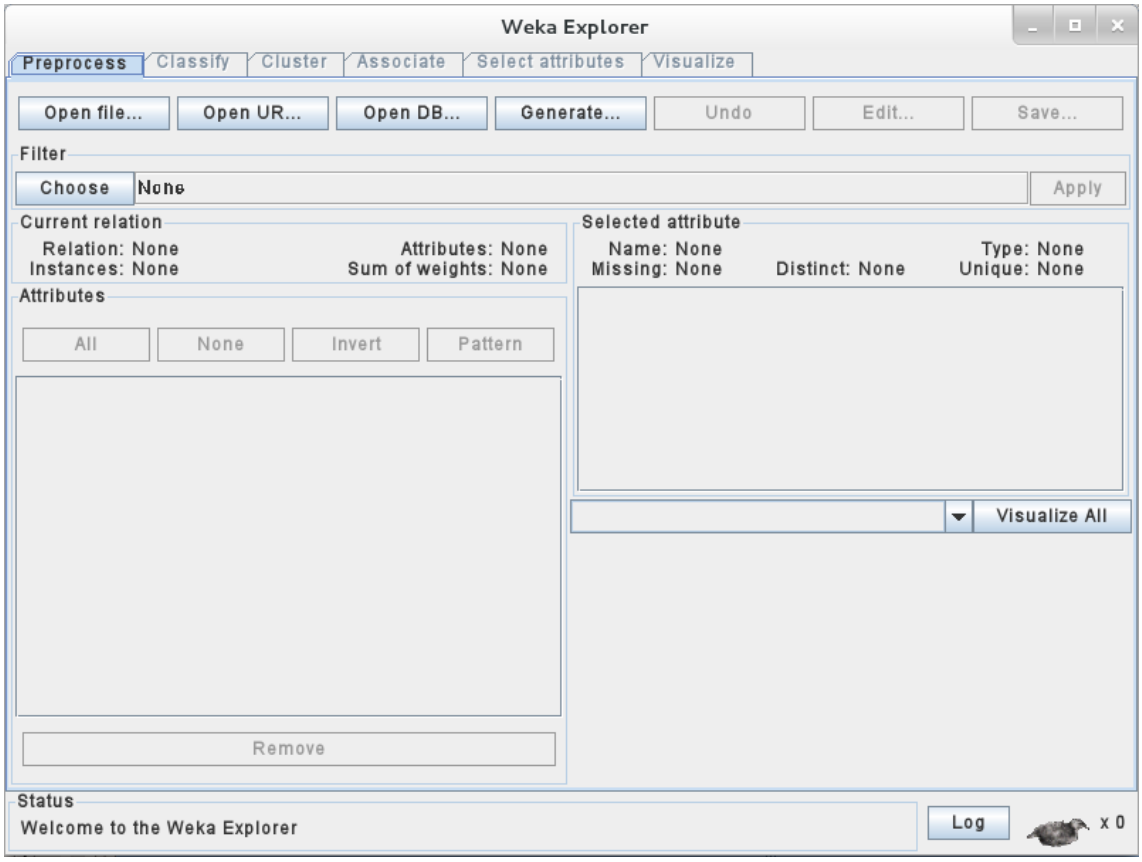

Figura 5 3: Pestaña “Preprocess” del explorador.

Una vez hecho esto, la pestaña *Preprocess* mostrará un resumen del conjunto de datos seleccionado en la sección *Current relation*, y un listado de atributos en la sección *Attributes* donde éstos se pueden seleccionar para que la sección *Selected attribute* muestre más detalles de algún atributo en particular como, por ejemplo, el tipo de atributo -numérico o nominal-, el porcentaje de valores *missing*, etc. (Figura 5 4).

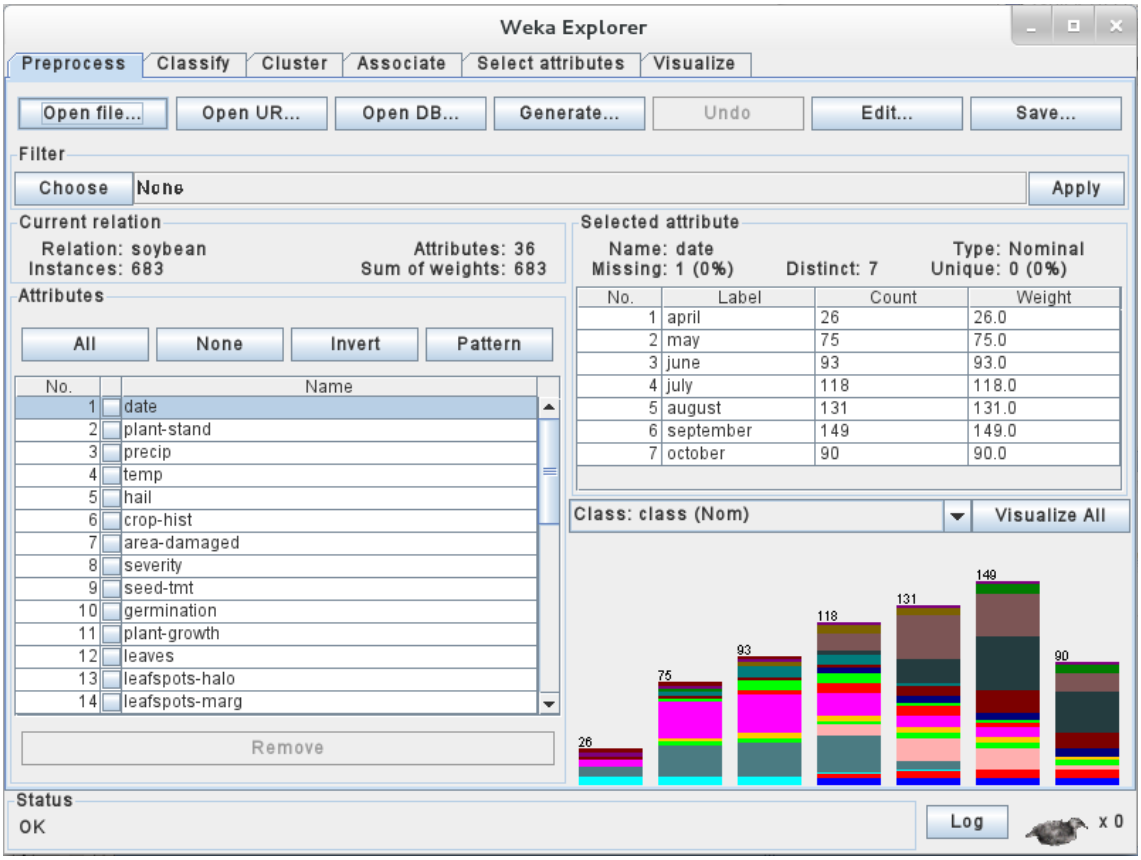

Figura 5 4: Pestaña “Preprocess” del explorador con la base de datos “soybean.arff” cargada.

Un ejemplo de la manipulación que se puede efectuar sobre los datos cargados se encuentra precisamente en esta sección *Selected attributes*, que permite eliminar por completo uno o varios atributos de la base de datos mediante la opción “*Remove*”.

Otra manera de manipular los datos es mediante la aplicación de filtros. Pulsando el botón *Choose* de la sección *Filter* se desplegará un listado de filtros, clasificados como “*supervised*” y “*unsupervised*”, y dentro de estas categorías encontraremos filtros de tipo “*attribute*” e “*instance*”.

Entre los numerosos filtros que ofrece Weka, mencionaremos algunos ejemplos que se han utilizado en este proyecto y sus aplicaciones:

Uno de estos ejemplos es el filtro ***ReplaceMissingValues***, que puede encontrarse dentro de los filtros de atributo no supervisados. Como su propio nombre indica, el resultado de aplicar este filtro a un conjunto de datos será la sustitución de los valores *missing* que existan en la base de datos por un valor concreto.

Esos valores con los que "rellena" los atributos *missing* salen de la media de los valores para dicho atributo -en el caso de atributos numéricos- o de la moda -en el caso de atributos nominales- dentro del *training set*.

La utilidad de este filtro en el proyecto ha residido en que, hasta terminar de implementar el tratamiento de los valores *missing* en JCHAID, muchas de las bases de datos disponibles para hacer pruebas no podrían utilizarse con este clasificador y sin embargo, una vez aplicado, pasan a ser bases de datos válidas.

Además, una vez implementado el tratamiento de *missing values*, también resultó muy útil para detectar errores en él; dado un error en la clasificación de una base de datos, si aplicando este filtro el error desaparecía, podía saberse con casi total seguridad que el error se encontraba en ese punto concreto.

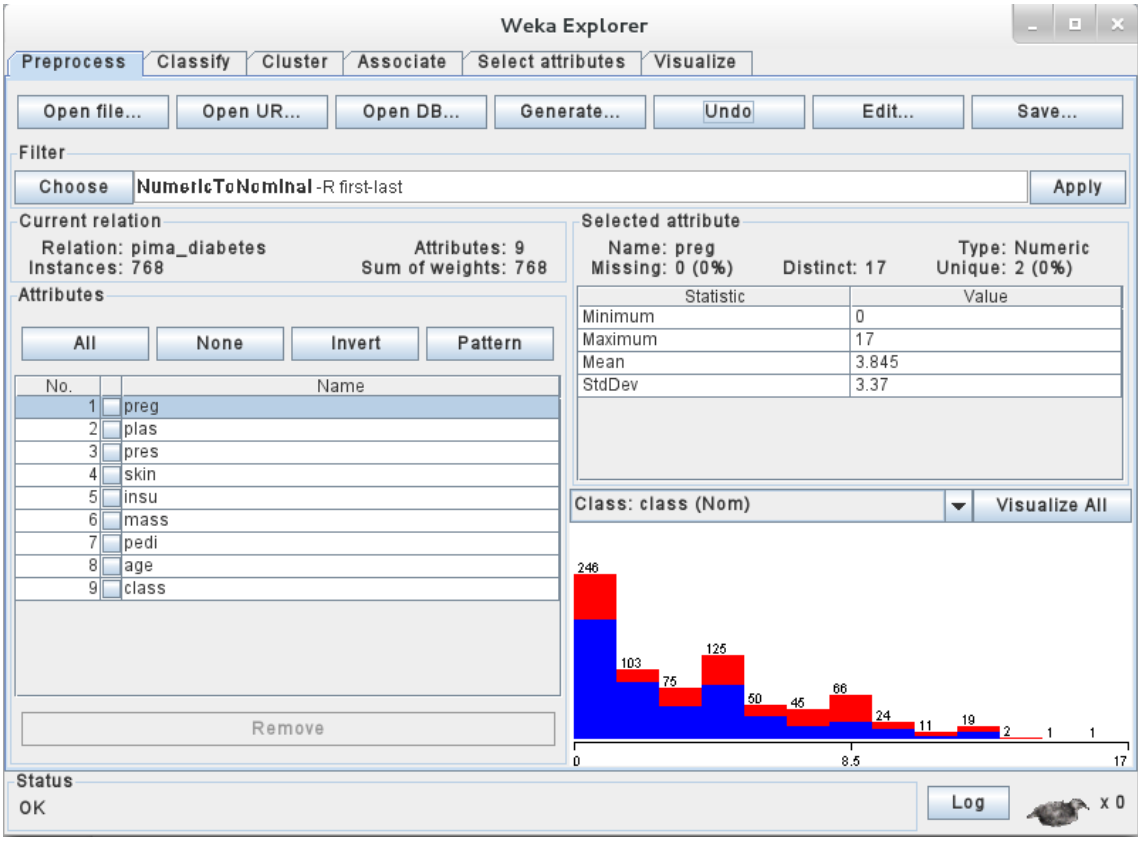

Figura 5 5: Atributo numérico antes de aplicar filtro NumericToNominal.

Otro ejemplo es ***NumericToNominal***, dentro de los filtros de atributo no supervisados. En este caso, el efecto de aplicar el filtro es el de “discretizar” los atributos numéricos de la base de datos, es decir, convertir todos los valores numéricos en nominales. Esto ha sido especialmente útil para este proyecto ya que, como el algoritmo CHAID no puede trabajar con atributos numéricos, la aplicación de este filtro permite utilizar el clasificador para muchas bases de datos con las que a priori no se podría.

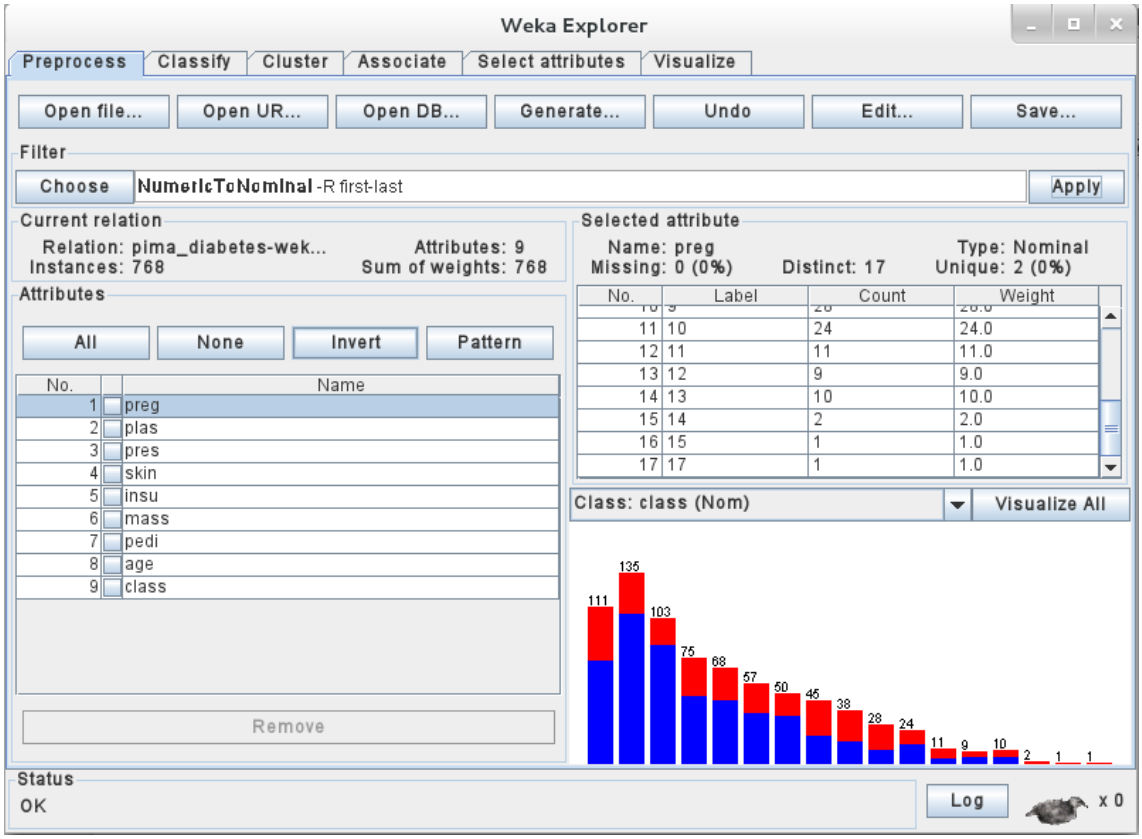

Figura 5 6: Atributo numérico tras aplicar filtro NumericToNominal.

Las figuras 5 5 y 5 6 muestran un atributo de ejemplo antes y después de aplicar el filtro *NumericToNominal*. Puede verse como la información que ofrece Weka cambia: primero indicaba "*Type: Numeric*" y después "*Type: Nominal*", y lo que primero era un valor continuo con rango de 0 a 17, después son 17 valores discretos.

Por último, merece la pena mencionar el filtro ***MergeNominalValues***, que se encuentra entre los filtros supervisados de atributo. En este caso, su importancia no reside en el uso que se le ha dado para preprocesar datos, sino en que la función de este filtro coincide con una parte importante del algoritmo CHAID y, por tanto, su código pudo ser reutilizado en gran medida para este proyecto. En el apartado 6.1.2 se explica más detalladamente el código fuente

de este filtro y su adaptación dentro de la implementación del algoritmo CHAID realizada para este proyecto.

5.3.1.2. Pestaña Classify

Hemos visto como cargar, visualizar y preprocesar un conjunto de datos. La pestaña *Classify* sirve para ejecutar un clasificador contra esos datos.

La sección *Classifier* permite elegir el clasificador a utilizar mediante el botón *Choose*. A su derecha aparecerá el nombre del clasificador elegido junto a sus parámetros. Inicialmente los parámetros mostrarán sus valores por defecto, pero se puede hacer *click* para abrir una ventana donde establecer otros valores.

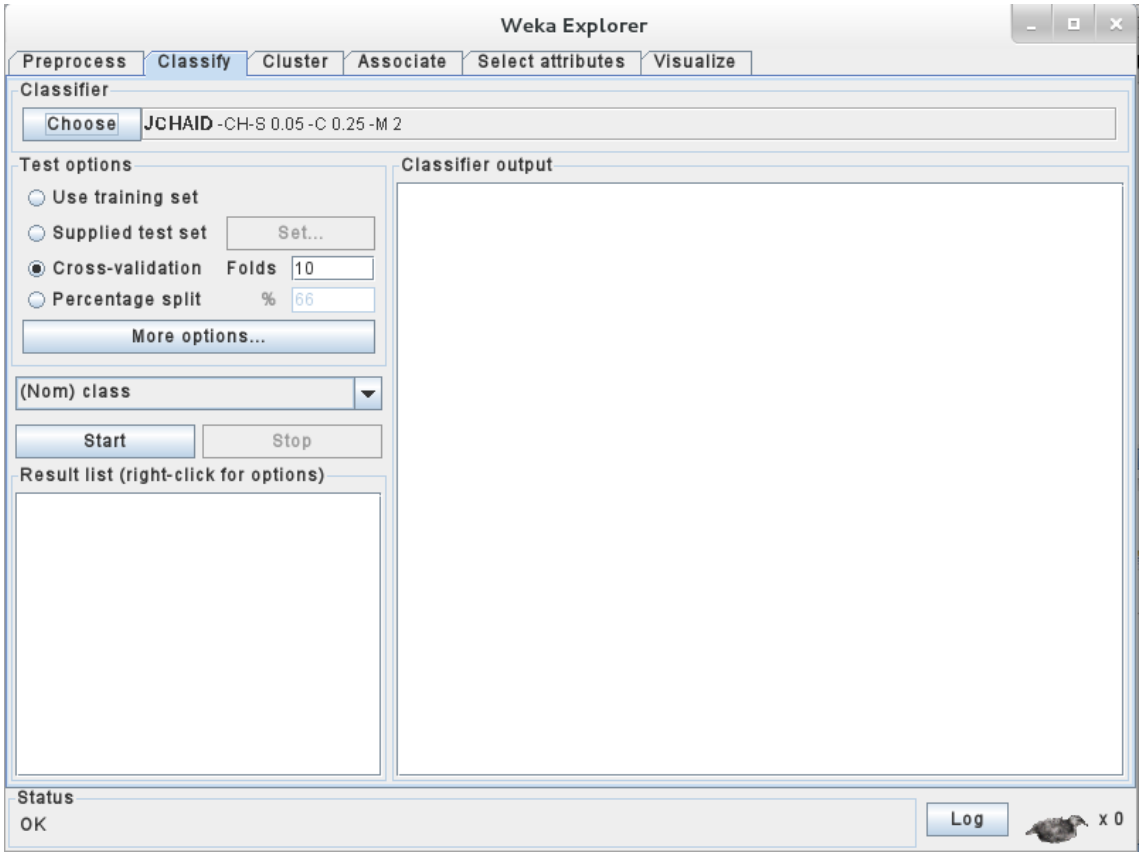

Figura 5 7: Pestaña “Classify” del explorador con el algoritmo JCHAID seleccionado.

La sección *Test options* permite elegir entre cuatro modelos de evaluación:

- *Use training set*: utiliza el mismo conjunto de datos como conjunto de entrenamiento y conjunto de validación.

- *Supplied test set*: permite seleccionar un conjunto de datos diferente para la validación. Esta opción ha resultado útil para este proyecto al permitir forzar situaciones concretas que resultara interesante probar. Ejemplo: ¿qué ocurre si un determinado valor que no estaba en el conjunto de entrenamiento aparece en el conjunto de validación? Como se explicará en el apartado 6.1.2, esto ha sido un caso a tener en cuenta

para evitar la aparición de errores en tiempo de ejecución.

- *Cross-validation*: explicada en el apartado 3.3.2.2 de esta memoria, esta opción realizará una validación cruzada utilizando el número de *folds* seleccionados.

- *Percentage split*: dividirá la base de datos en dos partes. Una parte contendrá el porcentaje elegido de todas las instancias del conjunto de datos, y lo usará como conjunto de entrenamiento. Las instancias restantes se usarán para la validación.

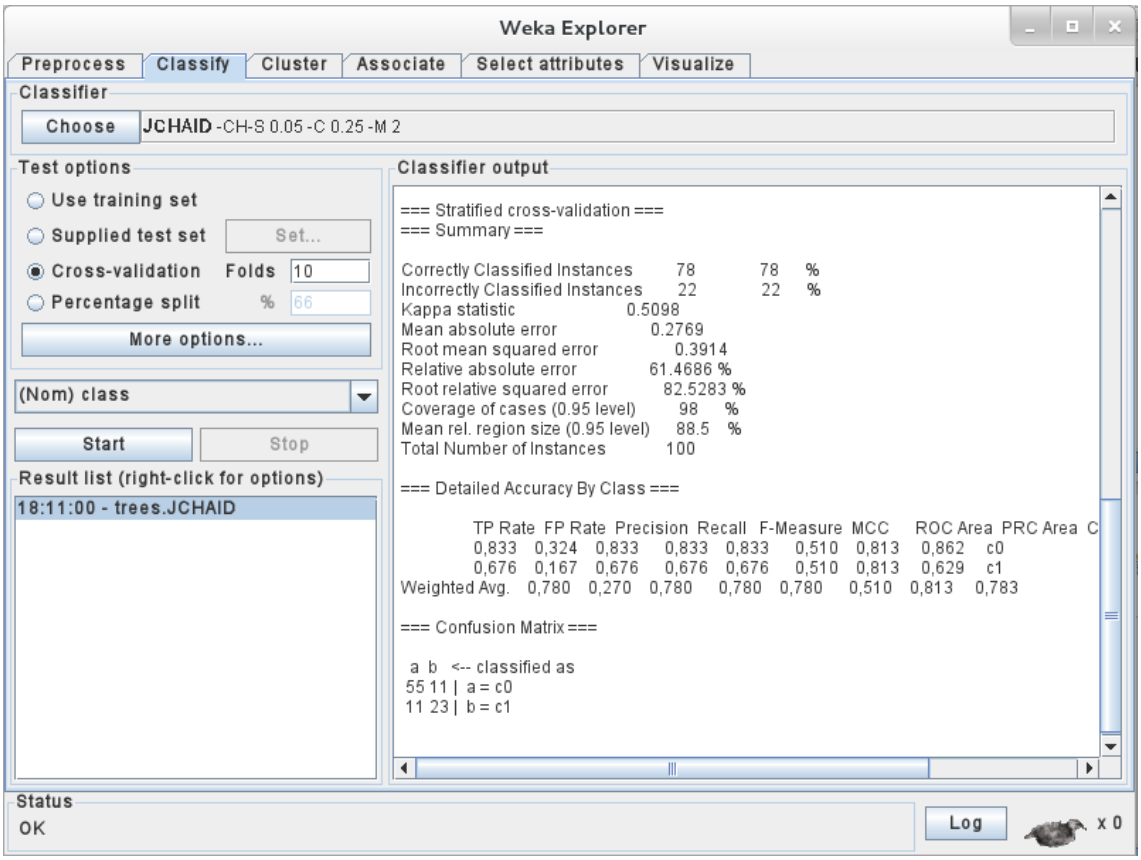

Figura 5 8: Pestaña “Classify” del explorador con los resultados de una clasificación.

El botón *Start* lanza la ejecución del algoritmo. En primer lugar, usará el conjunto de entrenamiento para generar el clasificador. Después, usará el conjunto de validación para probar el modelo generado. Finalmente imprimirá un resumen en la sección *Classifier output* donde podrá visualizarse tanto el modelo generado, como los resultados obtenidos en la fase de validación.

5.3.2. Experimenter

La aplicación Experimenter es el entorno de *benchmarking* para clasificadores incluído en Weka. Esto significa que sirve para automatizar la ejecución de uno o varios clasificadores contra uno o varios conjuntos de datos, ofreciendo al final distintas estadísticas de los resultados que se pueden analizar. Consta de las siguientes pestañas:

5.3.2.1. Pestaña Setup

En esta pestaña se configura el Experimento a realizar. Aunque se puede empezar a configurar desde cero mediante el botón *New*, el botón *Open* permite cargar configuraciones creadas previamente. Asímismo, el botón *Save* permite guardar la configuración en curso.

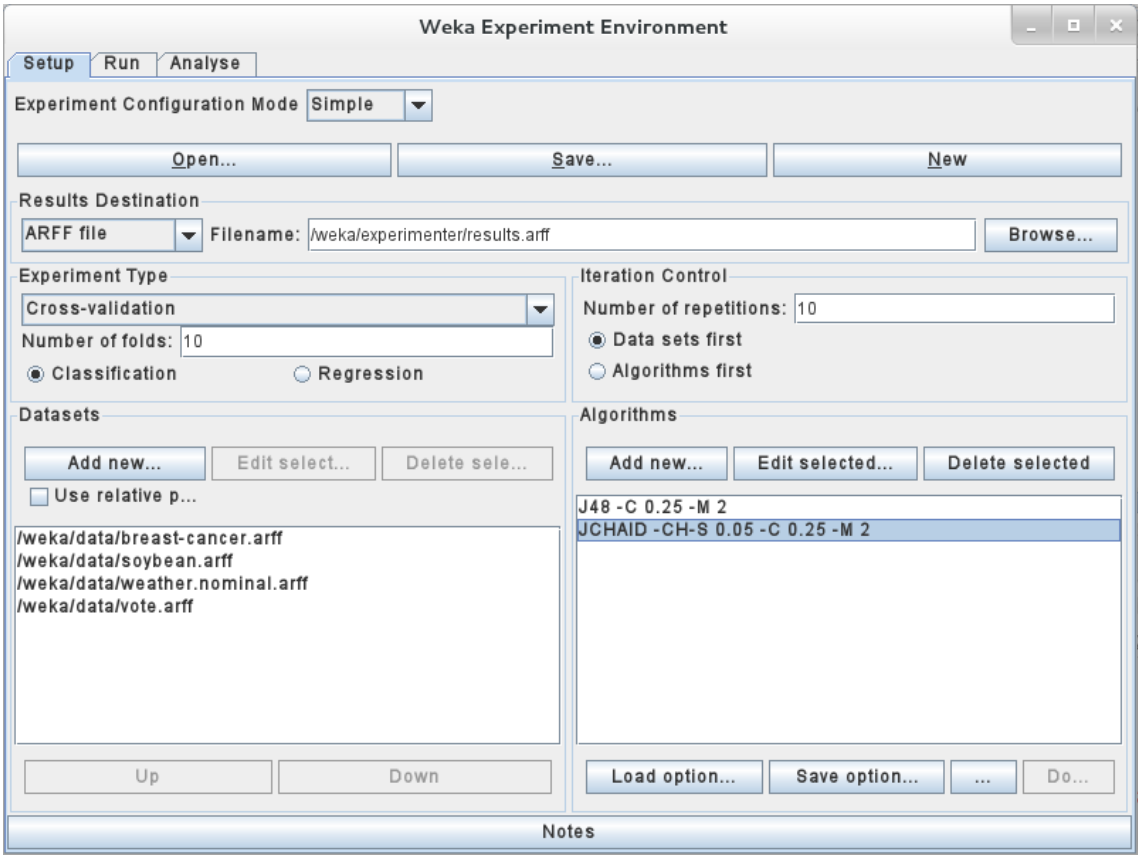

Figura 5 9: Pestaña “Setup” del experimentador.

En la Figura 5 9 pueden observarse dos grandes bloques: el de la izquierda sirve para seleccionar las bases de datos y la manera en que se usarán; el de la derecha sirve para seleccionar los algoritmos a ejecutar, sus opciones y el número de ejecuciones a realizar.

5.3.2.2. Pestaña Run

Esta pestaña es donde se pone a ejecutar el experimento configurado. Tras pulsar *Start*, una ventana de *Log* irá imprimiendo mensajes de información sobre la prueba que se está realizando.

Si no ocurren errores, este *Log* será tan escueto como el que aparece en la Figura 5 10.

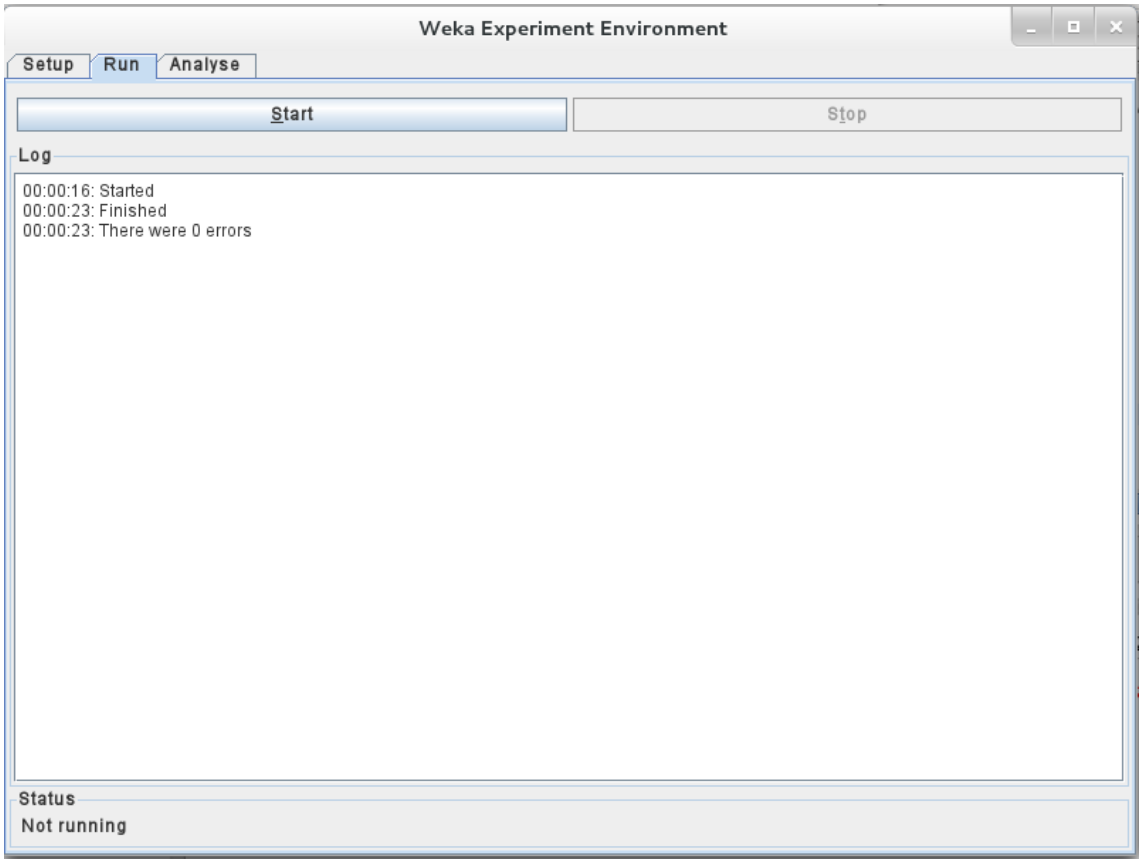

Figura 5 10: Pestaña “Run” del experimentador.

El botón *Stop* puede utilizar para detener la ejecución, por ejemplo, en caso de que el proceso esté tardando en terminar más de lo deseado y quiera probarse a cambiar algún parámetro de la configuración, como el número de bases de datos a utilizar, o el número de validaciones a aplicar.

5.3.2.3. Pestaña Analyse

Finalmente está la pestaña en la que se pueden visualizar los resultados del experimento, de la manera en que se muestra en la figura 5 11.

En primer lugar, la sección *Source* permite elegir los resultados que se quieren analizar -que pueden provenir del Experimento recién ejecutado -botón *Experiment*- o de algún otro lanzado en anteriores ocasiones y cuyos resultados se hayan exportado a un fichero o base de datos -botones *File* y *Database*, respectivamente-.

De entre las acciones que ofrece la sección *Actions*, la que interesa aquí es *Perform test*. Esto hará que se ejecute un test sobre los resultados seleccionados en *Source*, e imprimirá el resultado en la sección *Test output*.

En la figura 5 11 puede observarse la configuración por defecto del test y la presentación de unos resultados para un ejemplo de cuatro bases de datos y dos algoritmos. Por defecto se comprobará la tasa de aciertos de cada algoritmo para cada base de datos, y se presentará en formato de tabla.

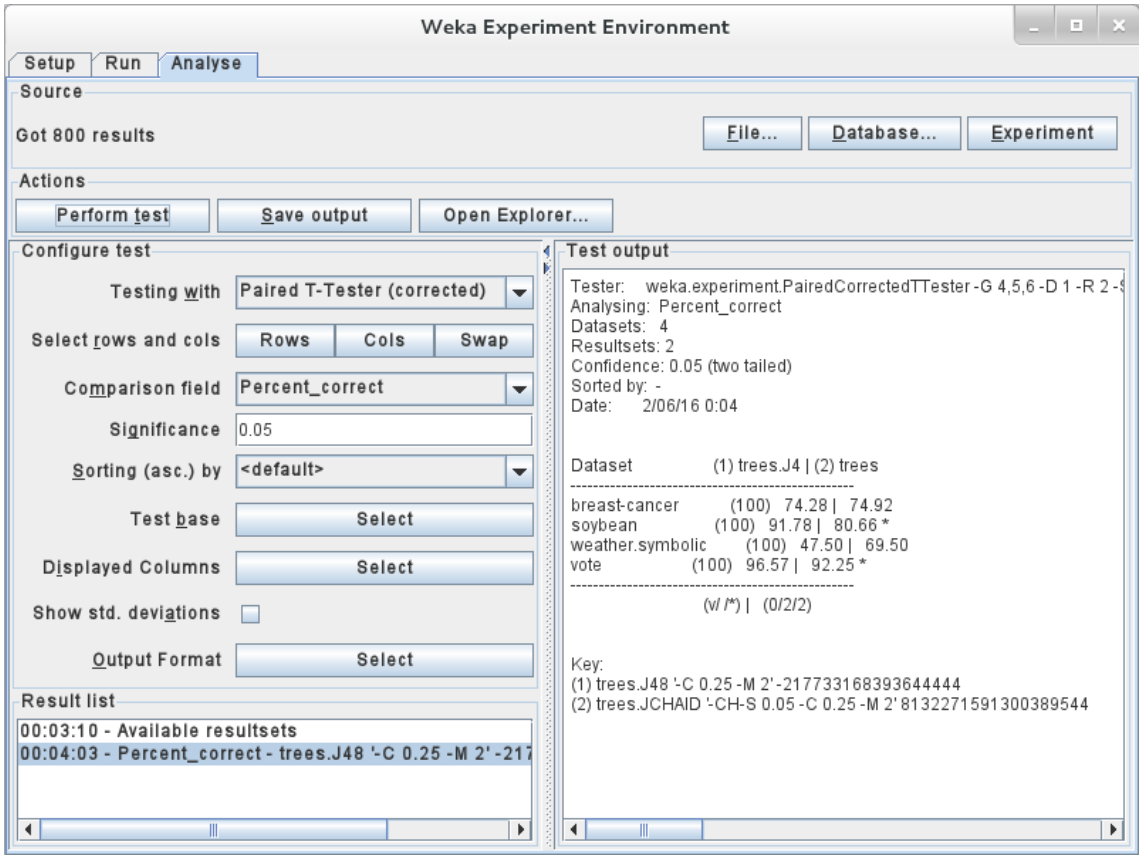

Figura 5 11: Pestaña “Analyse” del experimentador.

Pero la configuración puede ser modificada para realizar muchas más verificaciones sobre los resultados del experimento, así como el modo en que se prefieren presentar, si bien no es el objetivo de esta memoria entrar en mayor nivel de detalle.

5.4. Formatos de datos

Como se ha mencionado anteriormente, Weka es capaz de cargar conjuntos de datos de diferentes fuentes, principalmente bases de datos relacionales y ficheros en diferentes formatos. En este apartado se explicarán dos formatos en particular: el formato .ARFF, que es el más utilizado, y el .XRFF, menos común pero que ha tenido cierta utilidad en este proyecto.

5.4.1. Ficheros .ARFF

El nombre ARFF proviene de *Attribute Relation File Format*, y consiste en un fichero con contenido en texto plano, que representa una colección de datos siguiendo la siguiente estructura::

- Cabecera
    - Nombre de la colección
    - Listado de atributos
- Cuerpo
    - Listado de instancias

Nombre de la colección

Declarado mediante el token *@relation*, seguido del nombre. Esta propiedad da nombre a la base de datos, es decir, al problema de clasificación a resolver, y será el nombre que Weka mostrará en su interfaz de usuario.

Ejemplo:
@relation mi_coleccion

Listado de atributos

Cada atributo se declara con el token *@attribute* seguido del nombre de atributo y su tipo: cuando es numérico, el tipo es *numeric*. Cuando es nominal, no se indica un tipo, sino el conjunto de posibles valores, entre corchetes y separados por comas.

Ejemplo:
@attribute mi_atributo_numerico numeric
@attribute mi_atributo_nominal {valor1, valor2, valor3}

Listado de instancias

El listado comienza con el token *@data*, y se asume que lo que queda entre la siguiente línea hasta el final del fichero son las instancias de la colección. El formato de este bloque sigue el estilo de un fichero CSV: cada instancia ocupa una línea, y los valores de sus atributos se separan por comas.

Ejemplo:
@data
15, valor1
12, valor3
18, valor2

Ejemplo real: base de datos weather.arff

```
@relation weather

@attribute outlook {sunny, overcast, rainy}
@attribute temperature numeric
@attribute humidity numeric
@attribute windy {TRUE, FALSE}
@attribute play {yes, no}

@data
sunny,85,85,FALSE,no
sunny,80,90,TRUE,no
overcast,83,86,FALSE,yes
rainy,70,96,FALSE,yes
rainy,68,80,FALSE,yes
rainy,65,70,TRUE,no
overcast,64,65,TRUE,yes
sunny,72,95,FALSE,no
sunny,69,70,FALSE,yes
rainy,75,80,FALSE,yes
sunny,75,70,TRUE,yes
overcast,72,90,TRUE,yes
overcast,81,75,FALSE,yes
rainy,71,91,TRUE,no
```

Tabla 5.1: Contenido de la base de datos Weather en formato .arff

5.4.2. Ficheros XRFF

El nombre XRFF proviene de *eXtensible attribute-Relation File Format*, y se basa en el formato XML.

Nombre de la relación

Se declara en el atributo "*name*" del elemento "*<dataset>*"

Ejemplo:

<dataset name="weather">

[...]

</dataset>

Listado de atributos

Se enumeran en un listado acotado por las etiquetas <*attributes*> y <*/attributes*>. En el siguiente ejemplo se muestra la forma en que un XRFF declararía los mismos que en el ejemplo expuesto anteriormente para ficheros ARFF.

Ejemplo:

```
<attributes>
   <attribute name="mi_atributo_numerico" type="numeric"/>
   <attribute name="mi_atributo_nominal" type="nominal">
      <labels>
         <label>valor1</label>
         <label>valor2</label>
         <label>valor3</label>
      </labels>
   </attribute>
</attributes>
```

Listado de instancias

Se enumeran en un listado acotado por las etiquetas <*instances*> y <*/instances*>. Cada instancia es un elemento <*instance*> que a su vez contiene un elemento <*value*> por cada atributo de la base de datos,

```
<instances>
   <instance>
      <value>15</value>
      <value>valor1</value>
    </instance>
   <instance>
```

```
      <value>12</value>
      <value>valor3</value>
    </instance>
    <instance>
      <value>18</value>
      <value>valor2</value>
    </instance>
    [...]
</instances>
```

Ejemplo real: base de datos weather.xrff

```
<?xml version="1.0" encoding="utf-8"?>

<dataset name="weather" version="3.7.13">
  <header>
    <attributes>
      <attribute name="outlook" type="nominal">
        <labels>
          <label>sunny</label>
          <label>overcast</label>
          <label>rainy</label>
        </labels>
      </attribute>
      <attribute name="temperature" type="numeric"/>
      <attribute name="humidity" type="numeric"/>
      <attribute name="windy" type="nominal">
        <labels>
          <label>TRUE</label>
          <label>FALSE</label>
        </labels>
      </attribute>
      <attribute class="yes" name="play" type="nominal">
        <labels>
          <label>yes</label>
          <label>no</label>
        </labels>
      </attribute>
    </attributes>
  </header>
  <body>
    <instances>
      <instance>
        <value>sunny</value>
        <value>85</value>
        <value>85</value>
        <value>FALSE</value>
        <value>no</value>
      </instance>

[...]

      <instance>
        <value>rainy</value>
        <value>71</value>
        <value>91</value>
        <value>TRUE</value>
        <value>no</value>
      </instance>
    </instances>
  </body>
</dataset>
```

Tabla 5.2: Contenido de la base de datos Weather en formato .xrff

Para resumir se han quitado de esta muestra todas las instancias excepto la primera y la última -que se dejan a modo de ejemplo-, ya que el formato XML hace demasiado extensa la lista de atributos para mostrarla aquí al completo.

Esta versión de la base de datos *Weather* se ha generado con el propio Weka, que una vez tiene cargado un conjunto de datos permite exportarlos a otros formatos. En el elemento <*dataset*> puede verse que además del atributo *name*, explicado anteriormente, está el atributo *version="3.7.13"* indicando la versión de Weka con la que se ha exportado.

Caso especial: variables ordinales y metadatos

Un problema que surgió a la hora de implementar el algoritmo CHAID fue la imposibilidad de declarar variables de tipo ordinal en Weka. Como se mencionaba en el apartado 3.1.2, otros algoritmos no distinguen este tipo de dato, pero CHAID sí lo hace, por lo que era importante poder tener bases de datos con atributos ordinales.

Se planteó este problema en la lista de correo de desarrolladores del Weka, y la respuesta recibida por parte de Eibe Frank -uno de los creadores del Weka- proponía usar los metadatos XML de los ficheros XRFF para marcar las variables ordinales mediante la propiedad "*ordering*".

Un ejemplo de cómo definir una variable señalando que es ordinal mediante metadatos, sería el mostrado en la tabla 5.3:

```
 <attribute name="Class" type="nominal">
<labels>
  <label>0</label>
  <label>1</label>
</labels>
<metadata>
  <property name="ordering">ordered</property>
</metadata>
```

Tabla 5.3: Definición de variables ordinales en ficheros XRFF.

Al establecer la propiedad "*ordering*" como "*ordered*" se está haciendo saber al clasificador con el que se trabaje que la variable es ordinal. La forma de leer estos metadatos por parte del clasificador se explica más detalladamente en el apartado 6.2.1.1.

6. IMPLEMENTACIÓN

6.1. Extendiendo Weka

6.1.1. Añadiendo nuevos clasificadores

Para saber cómo se implementa un nuevo algoritmo en Weka, uno puede fijarse en cómo están organizados el resto de algoritmos existentes y las interfaces que éstos implementan, explicadas en [Bou-16].

En la figura 6 1 se observa cómo Weka divide el código en paquetes para los distintos tipos de algoritmos y componentes que puede integrar: *associations*, *classifiers*, *estimators*, *filters*… Para el caso de JCHAID y JCHAIDStar el paquete que interesa explorar es *weka.classifiers.trees*, que engloba todos los algoritmos para generar árboles de clasificación.

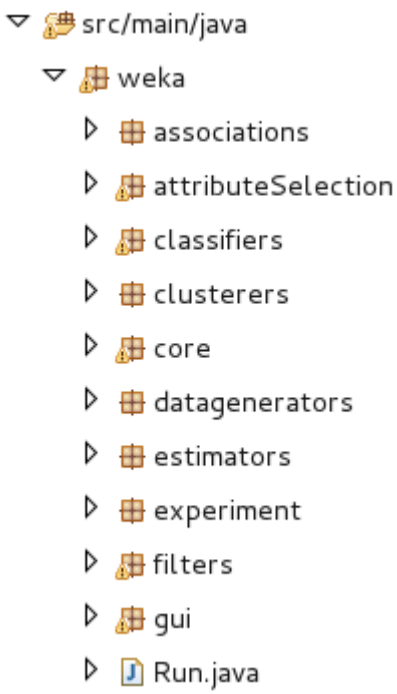

Figura 6 1: Paquetes de Weka.

Dentro de dicho paquete se encuentra el clasificador J48, que es el nombre de la implementación del C4.5, explicado en el apartado 4.2.

La clase J48 extiende AbstractClassifier, que a su vez implementa la interfaz Classifier.

Empezando por lo más básico, la interfaz Classifier expone los cuatro métodos básicos que debe implementar cualquier nuevo *clasificador simple* -existen otras interfaces para *meta-clasificadores* [Bou-16], pero no interesa en este caso- que se implemente en Weka:

```
public abstract void buildClassifier(Instances data) throws Exception;

public double classifyInstance(Instance instance) throws Exception;
```

```
public double[] distributionForInstance(Instance instance) throws Exception;

public Capabilities getCapabilities();
```

Un algoritmo que implemente esta interfaz, mediante estos cuatro métodos ya será capaz de construir un árbol clasificador -con *buildClassifier()*- y usarlo para clasificar nuevos casos -con *classifyInstance()* y *distributionForInstance()*-. Además, con *getCapabilities()* será capaz de indicar a la interfaz de usuario de Weka las "capacidades" del algoritmo, como por ejemplo los tipos de variables con los que es capaz de trabajar, etc.

La clase abstracta AbstractClassifier que implementa esta interfaz, además de estos cuatro métodos, añade unos cuantos dirigidos sobre todo a integrar el algoritmo con la interfaz de usuario.

A modo de ejemplo, a continuación se muestran los tres métodos dirigidos a mostrar el listado de opciones o parámetros que admite un clasificador, obtener sus valores y cambiarlos. Para resumir, solo se muestran las partes que manipulan el parámetro *output-debug-info*.

```
public Enumeration<Option> listOptions() {

  Vector<Option> newVector =
    Option.listOptionsForClassHierarchy(this.getClass(), AbstractClassifier.class);

  newVector.addElement(new Option(
    "\tIf set, classifier is run in debug mode and\n"
    + "\tmay output additional info to the console", "output-debug-info",
    0, "-output-debug-info"));
```

```
   [...]

   return newVector.elements();
}

public String[] getOptions() {

   Vector<String> options = new Vector<String>();
   for (String s : Option.getOptionsForHierarchy(this, AbstractClassifier.class)) {
      options.add(s);
   }

   if (getDebug()) {
      options.add("-output-debug-info");
   }

   [...]

   return options.toArray(new String[0]);
}

public void setOptions(String[] options) throws Exception {

   Vector<String> options = new Vector<String>();
   for (String s : Option.getOptionsForHierarchy(this, AbstractClassifier.class)) {
      options.add(s);
   }

   if (getDebug()) {
      options.add("-output-debug-info");
   }

   [...]
```

```
    return options.toArray(new String[0]);
}
```

Además, por cada parámetro que se añada al listado de *options* de esta manera, habrá que definir una función *getParameter()*, una *setParameter()* y un *parameterTipText()*. Siguiendo con el ejemplo del parámetro *debug*, sería así:

```
/**
 * Get whether debugging is turned on.
 */
public boolean getDebug() {
    return m_Debug;
}

/**
 * Set debugging mode.
 */
public void setDebug(boolean debug) {
    m_Debug = debug;
}

/**
* Returns the tip text for this property
*/
public String debugTipText() {
    return "If set to true, classifier may output additional info to the console.";
}
```

Todos los parámetros que se definan siguiendo estas reglas, aparecerán automáticamente en la interfaz de usuario de Weka al abrir las opciones del clasificador y se podrán manipular a través de ésta.

Por tanto, se ha visto como implementando ciertas interfaces y clases abstractas que expone Weka para añadir nuevos algoritmos, éstos quedan perfectamente integrados en la plataforma y su interfaz de usuario.

6.1.2. Adaptando el filtro MergeNominalValues

En el apartado 5.3.1.1 se ha mencionado el filtro MergeNominalValues y el hecho de que parte de su código fuente pudo aprovecharse para realizar esta implementación de CHAID. Según su documentación, se trata de un filtro que agrupa valores de atributos nominales basándose en el método CHAID descrito en [Kas-80].

Al encontrar este código, en un primer momento se consideró que la implementación de JCHAID podría verse simplificada hasta el punto de solamente tener que realizar una llamada a su método *mergeValues()*, con lo cual el grueso de este nuevo clasificador quedaría delegado en un método que forma parte del código base de Weka.

Sin embargo, analizando el filtro en mayor profundidad se detectaron algunos inconvenientes. Algunos tenían que ver con diferencias encontradas entre lo que hacía el código de la función *mergeValues()* y el algoritmo CHAID completo tal como se describe en [Kas-80].

Por ejemplo, el corrector de Bonferroni solo se aplica usando la ecuación 3.2 de [Kas-80], cuando ante una muestra con *missing values* habría que usar la ecuación 3.3. De hecho, el propio tratamiento que hace de los *missings values* es diferente al original.

Otro inconveniente importante fue que la visibilidad del método -declarado como *protected*- no ofrecía posibilidad de realizar la llamada desde clases

externas como JCHAID. El hecho de ser un filtro significa que extiende ciertas clases abstractas (*SimpleBatchFilter*, *SimpleFilter*, y *Filter*) que exponen su funcionalidad a la interfaz de usuario por medio de un conjunto de funciones públicas concretas -como la función estática *useFilter()* de *Filter*-. Estas funciones se encargan internamente de llamar a los métodos protegidos de los filtros que las extienden.

Por otro lado, el filtro no tiene en cuenta una casuística particular: la de que un atributo presentara categorías con cero casos en la muestra de entrenamiento, haciendo que el árbol construido no fuera capaz de clasificar estos valores. Esto, aunque poco frecuente, se vio que era posible y que podía dar lugar a errores en tiempo de ejecución, por lo que se tuvo que invertir bastante esfuerzo en detectarlo y solucionarlo.

En conclusión, no se podían reutilizar estas funciones sin hacer algunas modificaciones al código base de Weka. Esto era algo que se quiso evitar, porque el propósito en este proyecto era desarrollar un clasificador siguiendo "las reglas" definidas por Weka, y no modificar el código base de Weka para amoldarlo a las necesidades del clasificador.

En cualquier caso, es justo decir que este filtro sirvió como punto de partida para empezar a implementar el algoritmo CHAID.

6.1.3. Reutilizando cálculos para Bonferroni

Como se ha dicho en el apartado anterior, el filtro MergeNominalValues implementaba un cálculo del corrector de Bonferroni correspondiente a la ecuación 3.2 expuesta en [Kas-80] y que se puede ver a continuación:

```
protected double BFfactor(int c, int r) {
```

```
    double sum = 0;
    double multiplier = 1.0;
    for (int i = 0; i < r; i++) {
      sum += multiplier
        * Math
          .exp((c * Math.log(r - i) -
            (SpecialFunctions.lnFactorial(i) + SpecialFunctions.lnFactorial(r - i))));
      multiplier *= -1.0;
    }
    return sum;
}
```

Esta función se ha copiado en las clases de JCHAID encargadas de utilizar este cálculo -se explicarán en los apartados 6.2 y 6.3-, añadiendo además una implementación de las ecuaciones 3.1 y 3.3 del mismo autor.

6.1.4. Reutilizando cálculos para χ^2

Para todo lo relacionado con el estadístico χ^2, se encontró que los cálculos ya estaban implementados en las clases ContingencyTables y Statistics, ya existentes en Weka, por lo que allá donde JCHAID y JCHAIDStar los necesitan, sencillamente se llama a estas funciones:

Funciones de la clase ContingencyTables:

```
/**
 * Returns chi-squared probability for a given matrix.
 *
 * @param matrix the contigency table
 * @param yates is Yates' correction to be used?
```

```
 * @return the chi-squared probability
 */
public static double chiSquared(double [][] matrix, boolean yates) {

   int df = (matrix.length - 1) * (matrix[0].length - 1);

   return Statistics.chiSquaredProbability(chiVal(matrix, yates), df);
}

/**
 * Computes chi-squared statistic for a contingency table.
 *
 * @param matrix the contigency table
 * @param useYates is Yates' correction to be used?
 * @return the value of the chi-squared statistic
 */
public static double chiVal(double [][] matrix, boolean useYates) {

   [...]

}
```

Funciones de la clase Statistics:

```
/**
 * Returns chi-squared probability for given value and degrees of freedom.
 * (The probability that the chi-squared variate will be greater than x for
 * the given degrees of freedom.)
 *
 * @param x the value
 * @param v the number of degrees of freedom
 * @return the chi-squared probability
 */
```

```
public static double chiSquaredProbability(double x, double v) {

    if (x < 0.0 || v < 1.0) {
        return 0.0;
    }
    return incompleteGammaComplement(v / 2.0, x / 2.0);
}
```

6.2. JCHAID

6.2.1. Extendiendo J48

Para la implementación de CHAID se ha optado por partir del clasificador J48, que como se ha dicho, es el nombre de la implementación de C4.5 en Weka. El motivo es que ambas implementaciones van a tener muchos puntos en común; al fin y al cabo, para integrarse en la plataforma dentro del paquete de árboles de clasificación, deberán implementar las mismas interfaces base -*Classifier* y *AbstractClassifier* explicadas en el apartado 6.1-.

Por lo tanto habrá una buena proporción de código que será parecido o idéntico, especialmente las partes que no formen parte de la propia implementación del algoritmo CHAID y C4.5, respectivamente, sino que más bien estén relacionadas con la integración en Weka.

Un ejemplo de esto podrían ser las funciones para la visualización del árbol generado: no forman parte del algoritmo de construcción del árbol, sino que son funciones que Weka fuerza a implementar para poder mostrar los resultados a través de la interfaz de usuario. En este caso, aunque CHAID y C4.5 tengan formas diferentes de construir sus árboles de clasificación, después pueden compartir el código para visualizarlos porque en ambos casos se trata de la misma estructura de datos.

Por tanto, merece la pena plantear el JCHAID como una extensión de J48 donde solamente se sobreescriban aquellas partes que deban ser diferentes.

CHAID	J48
JCHAID	J48
CHAIDClassifierTree	C45PruneableClassifierTree
CHAIDDistribution	Distribution
CHAIDModelSelection	C45ModelSelection
CHAIDSplit	ClassifierSplitModel

Tabla 6.1: Correspondencia entre clases de JCHAID y clases de J48.

La tabla 6.1 enumera las clases que componen JCHAID y las clases equivalentes en J48, a las que sobreescriben. A continuación se explica la utilidad de cada una de ellas.

6.2.1.1. JCHAID

Extiende la clase J48, que a su vez extiende la clase AbstractClassifier: una implementación de la interfaz Classifier. AbstractClassifier y Classifier son parte del núcleo de Weka y la implementación de sus métodos es el primer paso para integrar un nuevo clasificador en esta plataforma.

De los métodos sobreescritos en esta clase, los más importantes son:

buildClassifier()

Este es el primer método de JCHAID al que Weka llamará para construir un árbol de clasificación. Es necesario sobreescribirlo para hacer que el árbol construído sea de tipo CHAIDClassifierTree, y no de tipo C45PruneableClassifierTree como haría el método original de J48. Se hace inicializando el nodo raíz del árbol, desde el cuál se construirán los demás, como se verá en CHAIDClassifierTree.

```
public void buildClassifier(Instances instances) throws Exception {

   [...]

   // Prepare the list of ordinal attributes, if exist
   prepareOrdinalAtts(instances);

   modSelection = new CHAIDModelSelection(
      m_minNumObj, instances, m_useMDLcorrection,
      m_doNotMakeSplitPointActualValue, m_CHsigLevel, m_ordinalAtts);

      m_root = new CHAIDClassifierTree(
         modSelection, false, m_CF, false, !m_noCleanup, false);

   [...]

}
```

getCapabilities()

Se sobreescribe para desactivar la capacidad de trabajar con atributos numéricos, ya que el CHAID básico no puede hacerlo.

```
public Capabilities getCapabilities() {
   Capabilities result = super.getCapabilities();

   result.disable(Capability.NUMERIC_ATTRIBUTES);

   return result;
}
```

setOptions()

Establece el valor de los parámetros del algoritmo. En el caso de JCHAID se sobreescribe para establecer el valor de los parámetros *CHsigLevel* y *CHordinalAttributeIndices*.

Los siguientes métodos no sobreescriben nada, sino que son propios de JCHAID:

prepareOrdinalAtts()

Establece la lista de atributos de tipo ordinal, definidos bien sea mediante el parámetro *CHordinalAttributeIndices* o mediante los metadatos del fichero en caso de que sea de tipo XRFF, como se explicaba en el apartado 5.4.2. En caso de que se den ambos tipos de definiciones a la vez, *prepareOrdinalAtts()* da prioridad a la de los metadatos del fichero XRFF.

```
protected void prepareOrdinalAtts(Instances instances) {

  m_ordinalAtts.setUpper(instances.numAttributes() - 1);
  // Determine list of ordinal attributes (via XRFF file)
  String rangeListXRFF = "";
  for (int i_att=0; i_att < instances.numAttributes(); i_att++) {
    // if properties were supplied for this attribute (via XRFF),
    if(instances.attribute(i_att).getMetadata() != null) {
      if (!m_XRFFUsed) {
        m_XRFFUsed = true;
      }

      // we get the attribute's ordering set
      if (instances.attribute(i_att).ordering() == Attribute.ORDERING_ORDERED) {
        [...]
```

```
      }
    }
  }

  if (m_XRFFUsed) {
    System.out.println("XRFF format was used. CHordinalAttributeIndices (CH-O)
option will be ignored!");
    [...]
  }
 }
```

6.2.1.2. CHAIDClassifierTree

Extiende la clase C45PruneableClassifierTree y ésta a su vez a ClassifierTree, también del paquete J48. Representa la estructura del árbol de clasificación: en particular, cada objeto CHAIDClassifierTree representa un nodo concreto del árbol.

De los métodos sobreescritos en esta clase, los más relevantes son los siguientes:

buildClassifier() & buildTree()

Estos son los métodos de construcción del árbol. Mediante llamadas recursivas, van generando los nodos del árbol uno a uno, desde el nodo raíz inicializado por JCHAID, hasta llegar a las hojas.

A continuación puede observarse como buildClassifier() inicia el proceso:

```
public void buildClassifier(Instances data) throws Exception {
```

```
   [...]

   buildTree(data, !m_cleanup);

   [...]
}
```

Y buildTree() continua de forma recursiva hasta llegar a los nodos hoja:

```
public void buildTree(Instances data, boolean keepData) throws Exception {

   [...]

   m_localModel = m_toSelectModel.selectModel(data);
   if (m_localModel.numSubsets() > 1) {
      localInstances = m_localModel.split(data);
      data = null;
      m_sons = new CHAIDClassifierTree[m_localModel.numSubsets()];
      for (int i = 0; i < m_sons.length; i++) {
         m_sons[i] = getNewTree(localInstances[i]);
         localInstances[i] = null;
      }
   } else {
      m_isLeaf = true;
      if (Utils.eq(data.sumOfWeights(), 0)) {
         m_isEmpty = true;
      }
      data = null;
  }
}
```

classifyInstance()

Método para clasificar casos una vez construído el árbol. Al igual que los dos métodos anteriores, funciona de manera recursiva: cada nodo valora en cuál de sus nodos hijos debe caer el caso a clasificar, hasta llegar a un nodo hoja. Este nodo hoja devolverá como resultado la clase en la que clasifica a ese caso. Esta información va volviendo hacia arriba, desde cada nodo a su nodo padre, hasta llegar al nodo raíz, que devuelve a Weka este resultado.

6.2.1.3. CHAIDModelSelection

Extiende C45ModelSelection y ésta a su vez extiende la clase abstracta ModelSelection, también del paquete J48. Se trata de la clase responsable de explorar las distintas posibilidades de división para cada nodo del árbol -es decir, los distintos atributos que se pueden usar como criterio de división- y elegir la mejor de todas.

Aquí se han extendido los siguientes métodos de ModelSelection:

CHAIDModelSelection()

Función constructora de la clase. Se sobreescribe para añadir los dos nuevos parámetros que acepta JCHAID, que como se ha explicado en su propio apartado son el nivel de significancia -sigLevel- y la lista de atributos ordinales -ordinalAtts-.

```
public CHAIDModelSelection(int minNoObj, Instances allData,
      boolean useMDLcorrection, boolean doNotMakeSplitPointActualValue,
      double sigLevel, Range ordinalAtts) {

   super(minNoObj, allData, useMDLcorrection, doNotMakeSplitPointActualValue);

   m_SigLevel = sigLevel;
   m_ordinalAtts = ordinalAtts;
}
```

selectModel()

Al igual que en J48, este método recorre la lista de atributos explorando las mejores divisiones posibles para cada uno de ellos. Después, compara entre sí estas divisiones candidatas para escoger la mejor. Si ésta se considera lo

suficientemente buena, se selecciona usa para dividir el nodo del árbol en que está trabajando; si no, sencillamente no se divide y se deja como nodo hoja.

```
public ClassifierSplitModel selectModel(Instances data) {

   [...]

   currentModel = new CHAIDSplit[data.numAttributes()];

   [...]

      // Get models for current attribute.
      currentModel[i] = new CHAIDSplit(i, m_minNoObj, sumOfWeights,
         m_useMDLcorrection, m_SigLevel, isOrdinalAtt(i));
      currentModel[i].buildClassifier(data);

   [...]

}
```

La importancia de sobreescribir este método reside en que las divisiones generadas sean de tipo CHAIDSplit, y no de tipo C45Split como ocurriría llamando al método padre. Además, debe cambiar el criterio por el que se selecciona una u otra división: mientras en J48 se usa el criterio de la entropía, aquí se utilizará el de χ^2.

Además se ha creado este nuevo método:

isOrdinalAtt()

Indica si un atributo dado es de tipo ordinal, usando el listado de atributos ordinales que ha recibido en el método constructor. Esta información se la pasa

a cada objeto CHAIDSplit generado -que se explica a continuación-, para que éste pueda adaptar su forma de calcular la mejor división para el atributo según de qué tipo sea: nominal (false) u ordinal (true).

```
public boolean isOrdinalAtt(int i_att) {
   return m_ordinalAtts.isInRange(i_att);
}
```

6.2.1.4. CHAIDSplit

Extiende C45Split y ésta a su vez la clase abstracta ClassifierSplitModel, también del paquete J48. Sirve para calcular y representar las posibles divisiones de un nodo dado. Si la clase CHAIDModelSelection se ha dicho que busca el mejor atributo por el que dividir un nodo, en CHAIDSplit lo que se elige es la división concreta de un atributo dado, es decir: el número de ramas en que se divide, y qué valores se asignan en cada rama.

Algunos de los métodos extendidos en esta clase son:

buildClassifier()

Este método forma parte del proceso de construcción del árbol. El método padre -el buildClassifier() de C45Split- comprobaba el tipo de atributo que tenía que dividir, y según fuera numérico o nominal llamaba al método handleEnumeratedAttribute() o handleNumericAttribute() respectivamente para calcular la división.

En este caso, ya que CHAID solo trabaja con atributos nominales se elimina dicha comprobación y cualquier llamada a handleNumericAttribute().

```
public void buildClassifier(Instances trainInstances) throws Exception {
```

```
    // Initialize the remaining instance variables.
    m_numSubsets = 0;

    // Only enumerated attributes.
    m_complexityIndex = trainInstances.attribute(m_attIndex).numValues();
    m_index = m_complexityIndex;
    m_missingsIdx = m_complexityIndex;
    handleEnumeratedAttribute(trainInstances);
}
```

handleEnumeratedAttribute()

Es el método que en J48 generaba las divisiones para los atributos de tipo nominal, que en el caso de CHAID es el único caso posible. Las mayores diferencias entre J48 y JCHAID comienzan en este punto.

Mientras que en J48, dado un atributo nominal, simplemente se genera un subnodo para cada categoría o valor, en JCHAID comenzarían las fases de unión y división de categorías -descritas en el apartado 4.3.1- para intentar buscar la mejor combinación posible.

Esta lógica de unión y división se ha implementado en la clase CHAIDDistribution mediante la función *mergeValues()* como se ve a continuación.

```
protected void handleEnumeratedAttribute(Instances trainInstances)
throws Exception {

    [...]
```

```
    // Paso 1: Obtener tabla de contingencia de todas las categorías del atributo
    // contra todas las clases de la variable dependiente
    m_distribution = new CHAIDDistribution(m_complexityIndex,
        trainInstances.numClasses(), m_minNoObj, m_SigLevel, m_ordered);

    [...]

    // Pasos 2-6 se ejecutan en mergeValues()
    getCHAIDDistribution().mergeValues();

    [...]
}
```

whichSubset()

Método clave de cara a clasificar los casos una vez se ha construído el árbol. Indica a cuál de los subnodos o ramas de la división pertenece un caso dado.

```
public int whichSubset(Instance instance) throws Exception {

    int originalCategoryIndex;

    if (instance.isMissing(m_attIndex)) {
        originalCategoryIndex = m_missingsIdx;
    } else {
        originalCategoryIndex = (int) instance.value(m_attIndex);
    }

    return getCHAIDDistribution().getSubsetIndex(originalCategoryIndex);
}
```

Ya que durante la construcción del árbol se ha pasado por el proceso de Unión en el que, probablemente, se han combinado algunas categorías, en este punto

CHAIDSplit no conoce exactamente a qué subnodo ha quedado asignado cada una.

Esta información se encuentra en el objeto m_distribution, de tipo CHAIDDistribution, y se accede a ella mediante el método getSubsetIndex() explicado en el siguiente apartado.

6.2.1.5. CHAIDDistribution

Extiende la clase Distribution de J48, donde simplemente representaba la tabla de contingencia para un nodo dado. Como ya se ha mencionado, en el caso de JCHAID contiene además toda la lógica de las fases de Unión/División de categorías del atributo donde reside la mayor diferencia del algoritmo CHAID respecto a otros.

Hay que tener en cuenta que esta lógica de Unión/División supone manipular la tabla de contingencia del nodo. Por ejemplo, si se unen dos categorías del atributo, en la tabla de contingencia hay que combinar también las filas que corresponden a esas dos categorías.

Sin embargo, cuando termine de construirse el árbol y empiece a utilizarse para clasificar, será necesario tener una relación entre las posiciones que ocupaban las categorías originalmente en la tabla con las posiciones que ocupan después de la fase de unión de categorías.

Para guardar este tipo de relaciones se han añadido a la clase tres vectores como atributos privados: "*m_indicators*", "*m_subsetIndicators*" y "*m_trainedIndicators*".

- El vector *m_indicators* es el que relaciona cada categoría original con el grupo en el que se haya metido tras la fase de Unión/División. Cada posición del vector corresponde a una de las categorías originales, y sus valores corresponden con los grupos de categorías resultantes. Este vector se usa durante la construcción del árbol.

- El vector *m_subsetIndicators* relaciona cada categoría del nodo padre, con el nodo hijo -en el código se llama *subsets* a los nodos hijo- al que se ha asignado la categoría tras la división del nodo padre. Este vector se usa para clasificar nuevos casos una vez construído el árbol.

- El atributo *m_trainedIndicators* es un vector de valores *booleanos* que indican si existen o no casos en la muestra de entrenamiento para cada una de las categorías originales. Se usa tanto en la fase de construcción del árbol como en la de clasificación, para evitar que se intenten realizar operaciones con categorías para los que no hay casos conocidos.

Los nuevos métodos de CHAIDDistribution respecto a Distribution son los siguientes:

mergeValues()

Es la principal función que realiza la fase de Unión explicada en el apartado 4.3.1.

A continuación se muestra una versión muy resumida del método, ya que por su complejidad es de los métodos más largos del clasificador. En cualquier caso, basta con los comentarios y las llamadas a funciones auxiliares para visualizar el flujo que seguiría el algoritmo para un conjunto de categorías dado. Los detalles de implementación que aquí se omiten pueden consultarse en el código fuente adjunto a este proyecto.

```
public void mergeValues() {

   […]

   // Can't merge further if only one subset remains
   while (this.numKnownTrainedBags() > 1) {
      // Paso 2: Calcular estadístico ChiCuadrado para cada pareja de categorías

      […]
      int toMergeOne = -1;
      int toMergeTwo = -1;
      […]

      // if no merge is possible
      if toMergeOne == -1)
         break;

      // Paso 3: Calcular el valor de probabilidad p de las parejas
      // para ver si nos queda alguna que no sea significativa (caso 1)
      // o si todas son significativas (caso 2)

      // Is least significant difference still significant?
      double chiSquaredProb = Statistics.chiSquaredProbability(
         minVal, reducedCounts[0].length - 1);
      if (Utils.gr(chiSquaredProb, m_SigLevel)) {
         // Paso 3 - Caso 1: Hemos encontrado un par cuyo valor p de probabilidad
         // es mayor al umbral. Por tanto el par con mayor valor p (es decir, el que
         // tiene menor valor ChiCuadrado) se mergeará y se irá al paso 4.
         // Replace matrix
         this.mergeCategories(toMergeOne, toMergeTwo);
         this.mergeIndicators(toMergeOne, toMergeTwo);
```

```
            [...]
        } else {
            // Paso 3 - Caso 2: Como todos los pares son significativos, hay que saltar al
paso 5.
            break;
        }
    }

    // Paso 5: Cualquier categoría (o grupo) con menor número de observaciones
(individuos o casos),
    // que el valor mínimo establecido se deberá unir con aquel grupo que sea más
parecido a él,
    // medido esto por el que tenga el 'pairwise chi-square' más pequeño.
    this.mergeSmallGroups();

    if (this.hasMissingValues()) {
        this.handleMissingValues();
    }

    // Paso 6: Calcular el valor de probabilidad ajustado para la combinación
    // de categorías definitiva.
    [...]
    m_chiSquaredProb = originalSig * bonferroniFactor( [....]  );
    [...]
}
```

mergeCategories()

De las mencionadas funciones auxiliares en las que se apoya *mergeValues()* para realizar la fase de Unión, quizá ésta sea la más importante por ser la que ejecuta la propia unión de categorías en la tabla de contingencia.

```
public void mergeCategories(int toMergeOne, int toMergeTwo) {

  // Reduce table by merging
  double[][] new_perClassPerBag = new double[m_perClassPerBag.length - 1][];
  for (int i = 0; i < m_perClassPerBag.length; i++) {
    if (i < toMergeTwo) {

      // Can simply copy reference
      new_perClassPerBag[i] = m_perClassPerBag[i];

    } else if (i == toMergeTwo) {

      // Need to add counts
      for (int k = 0; k < m_perClassPerBag[i].length; k++) {
        new_perClassPerBag[toMergeOne][k] += m_perClassPerBag[i][k];
      }

    } else {
      // Need to shift row
      new_perClassPerBag[i - 1] = m_perClassPerBag[i];
    }
  }

  m_perClassPerBag = new_perClassPerBag;

  m_perBag = new double[m_perClassPerBag.length];
  m_perClass = new double[m_perClassPerBag[0].length];
  totaL = 0;
  for (int i = 0; i < m_perClassPerBag.length; i++) {
    for (int j = 0; j < m_perClassPerBag[i].length; j++) {
      m_perBag[i] += m_perClassPerBag[i][j];
      m_perClass[j] += m_perClassPerBag[i][j];
      totaL += m_perClassPerBag[i][j];
```

```
    }
  }
}
```

getSubsetIndex()

Una vez terminada la fase de Unión, si la división a la que está asociada este objeto CHAIDDistribution termina siendo la escogida para dividir el nodo, necesitará conocer la posición que finalmente ocupa cada categoría entre los nodos hijos.

Para ello se ha añadido a la clase el método getSubsetIndex, que dada la posición original de una categoría indica en qué combinación de categorías se ha agrupado. Dicho de otra manera: a qué nodo hijo se le ha asignado.

```
public int getSubsetIndex(int originalCategoryIndex) {
  return m_subsetIndicators[originalCategoryIndex];
}
```

Como se ha explicado anteriormente, *m_subsetIndicators* es un vector que relaciona la posición original de cada categoría con la posición del nodo hijo en el que ha quedado agrupada.

6.3. JCHAIDStar

6.3.1. Extendiendo JCHAID

En el caso de JCHAID* la necesidad de heredar las clases de JCHAID es todavía más clara, al fin y al cabo el algoritmo en sí no es otra cosa que una extensión del original, como se explicó en el apartado 4.4.

Además, la única parte que lo diferencia frente al CHAID original es el soporte para atributos numéricos y el uso de la poda, características que se pueden recuperar del J48 con ciertas adaptaciones.

JCHAID*	JCHAID	J48
JCHAIDStar	JCHAID	J48
CHAIDStarClassifierTree	CHAIDClassifierTree	C45PruneableClassifierTree
CHAIDStarModelSelection	CHAIDModelSelection	C45ModelSelection
CHAIDStarSplit	CHAIDStarSplit	ClassifierSplitModel
ChiSquareSplitCrit		SplitCriterion

Tabla 6.2: Correspondencia entre clases de JCHAID* y clases de JCHAID y J48.

La tabla 6.2 indica la relación entre las clases de JCHAID*, del JCHAID original y de J48. A continuación se explican sus diferencias respecto a las que sobreescriben.

6.3.1.1. JCHAIDStar

Extiende la clase JCHAID, siendo la clase “raíz” de todo el algoritmo JCHAIDStar, al que Weka llamará para lanzar la construcción del clasificador.

JCHAIDStar()

Función constructora de la clase, aquí se usa para sobreescribir el valor por defecto del atributo *collapseTree* dejándolo desactivado.

```
public JCHAIDStar() {
   m_collapseTree = false;
}
```

Esto se debe a que *colapsar* es una opción de J48 que ALDAPA adaptó a CHAID* en la plataforma Haritza observando que daba mejores resultados en la experimentación pero, en este caso, la implementación de esta opción queda fuera del alcance del proyecto.

buildClassifier()

Como ocurría en JCHAID, es necesario sobreescribir este método para hacer que el árbol construído sea de tipo CHAIDStarClassifierTree, y no de tipo CHAIDClassifierTree como haría el método padre.

Además, en este caso se instancia la clase del árbol pasándole algunas opciones propias de J48 que en JCHAID se ignoraban: *Prune Tree*, *Subtree Raising*, y *Collapse Tree*.

```
public void buildClassifier(Instances instances) throws Exception {

   [...]

   modSelection = new CHAIDStarModelSelection(
      m_minNumObj, instances, m_useMDLcorrection,
      m_doNotMakeSplitPointActualValue, m_CHsigLevel, m_ordinalAtts);
```

```
    m_root = new CHAIDStarClassifierTree(
      modSelection, !m_unpruned, m_CF, m_subtreeRaising,
      !m_noCleanup, m_collapseTree);

    [...]

}
```

getCapabilities()

Se sobreescribe para volver a activar la capacidad de trabajar con atributos numéricos, ya que ésta es la diferencia principal -además de la poda- de CHAID* respecto al CHAID básico.

```
public Capabilities getCapabilities() {
    Capabilities result = super.getCapabilities();

    result.enable(Capability.NUMERIC_ATTRIBUTES);

    return result;
}
```

CHAIDStarClassifierTree

La clase encargada de construir y representar el árbol de clasificación, en este caso tiene la responsabilidad añadida de realizar la poda del árbol después de construirlo. Esta característica es, junto al tratamiento de variables continuas, la segunda diferencia de CHAID* respecto al CHAID original, como se explicó en el apartado 4.4.

buildClassifier()

Como se ha dicho anteriormente, este método es el que da comienzo a la construcción del árbol mediante una serie de llamadas recursivas a buildTree(), generando todos los nodos intermedios hasta llegar a las hojas.

Este es el método donde se añade un proceso de poda con la llamada al método prune().

El método collapse() existe como herencia de J48, pero como se ha explicado anteriormente, no está adaptada a CHAID* porque se trata de una funcionaldiad que queda fuera del alcance del proyecto. En cualquier caso, no se llegará a dar la llamada a esa función porque el parámetro *m_collapseTheTree* quedaba deshabilitado en el constructor de la clase JCHAIDStar.

```
public void buildClassifier(Instances data) throws Exception {

   [...]

   buildTree(data, m_subtreeRaising || !m_cleanup);
   if (m_collapseTheTree) {
      collapse();
   }
   if (m_pruneTheTree) {
      prune();
   }

   [...]
}
```

prune()

Sobreescribe el método prune() de J48 -en JCHAID no se usa para nada-, replicando la lógica del método padre para aplicarlo a las clases del JCHAIDStar.

Por ejemplo, este fragmento del método prune():

```
// Compute error for largest branch
Distribution dist = localModel().distribution();
indexOfLargestBranch = localModel().distribution().maxBag();
```

Pasa a ser así en JCHAIDStar:

```
// Compute error for largest branch
Distribution dist = localModel().distribution();
if (dist instanceof CHAIDDistribution) {
   indexOfLargestBranch = ((CHAIDDistribution)dist).maxActualBag();
} else {
   indexOfLargestBranch = dist.maxBag();
}
```

6.3.1.2. CHAIDStarModelSelection

En este caso solo se ha extendido el siguiente método de CHAIDModelSelection:

selectModel()

En este caso, aunque la lógica del método debe ser la misma en ambos algoritmos, es necesario sobreescribirlo para hacer que la división generada para el nodo se represente con un objeto de tipo CHAIDStarSplit, y no un CHAIDSplit como haría el método padre. Hay que tener en cuenta que la clase

CHAIDStarSplit es la que tendrá implementada la lógica para el tratamiento de atributos numéricos.

Aquí se muestran los únicos cambios que se hacen respecto al método padre:

```
public ClassifierSplitModel selectModel(Instances data) {

   […]

   currentModel = new CHAIDStarSplit[data.numAttributes()];

   […]

   // Get models for current attribute.
   currentModel[i] = new CHAIDStarSplit(
      i, m_minNoObj, sumOfWeights, m_useMDLcorrection,
      m_SigLevel, isOrdinalAtt(i));
   currentModel[i].buildClassifier(data);

   […]

}
```

6.3.1.3. CHAIDStarSplit

Como se ha explicado anteriormente, las dos diferencias fundamentales entre CHAID y CHAID* es que éste añade el tratamiento de atributos numéricos y el proceso de poda.

Si el proceso de poda se abordaba en la clase CHAIDStarClassifierTree, en CHAIDStarSplit se incluye el tratamiento de atributos numéricos, por lo que ésta será una de las clases que más cambios tiene respecto a la original.

Para ello se sobreescriben los siguientes métodos:

buildClassifier()

En CHAIDStarSplit se restaura la capacidad de tratar atributos numéricos eliminada en CHAIDSplit -ver apartado 6.2.1.4-. Para ello se vuelve a poner la condición que existía en J48 para distinguir entre nominales y numéricos:

```
public void buildClassifier(Instances trainInstances) throws Exception {

[...]

   // Different treatment for enumerated and numeric attributes.
   if (trainInstances.attribute(m_attIndex).isNominal()) {
      m_complexityIndex = trainInstances.attribute(m_attIndex).numValues();
      m_index = m_complexityIndex;
      handleEnumeratedAttribute(trainInstances);
   } else {
      m_complexityIndex = 2;
      m_index = 0;
      trainInstances.sort(trainInstances.attribute(m_attIndex));
      handleNumericAttribute(trainInstances);
   }
}
```

whichSubset()

Este método se vuelve a sobreescribir para restaurar su capacidad de clasificar atributos numéricos, que en CHAIDSplit se había eliminado. Para ello, el código del método queda así:

```
public final int whichSubset(Instance instance) throws Exception {

   if (instance.attribute(m_attIndex).isNominal()) {
      return super.whichSubset(instance);
   } else {
      if (instance.isMissing(m_attIndex)) {
         return m_missingCurrentIndex;
      } else if (Utils.smOrEq(instance.value(m_attIndex), m_splitPoint)) {
         return 0;
      } else {
         return 1;
      }
   }
}
```

Se diferencia del whichSubset() original de J48 en dos cosas:

La primera, que en el caso de llegar un valor *missing*, J48 devolvía "-1" indicando que el caso está repartido entre todos los subnodos, mientras que en JCHAID se encuentra en un subnodo particular señalado por el atributo *m_missingCurrentIndex*.

La segunda diferencia es que, en el caso de los atributos nominales, J48 asignaba un subnodo a cada categoría mientras JCHAID realiza el proceso de *Merging* ya explicado en el apartado 4.3. Por tanto, en este punto el trabajo se delega en el método padre -explicado en el apartado 6.2.1.4- que ya está preparado para detectar a qué combinación de categorías corresponde el caso que se está clasificando.

También se añade el siguiente método:

handleNumericAttribute()

Replicando el método original de J48, recorre el rango de posibles valores del atributo numérico en busca del mejor punto de división posible.

```
public void handleNumericAttribute(Instances trainInstances) throws Exception {

  [...]

  // In C4.5, only Instances with known values are relevant.

  [...]

  // In CHAID*, instances with unknown values are relevant too.
  if (hasMissings) {
     while (enu.hasMoreElements()) {
        instance = enu.nextElement();
        m_distribution.add(2, instance);
     }
  }

  [...]

  // Compute values of criteria for all possible split indices.
  while (next < firstMiss) {

     [...]

     if (Utils.gr(currentChiVal, m_chiSquaredVal)) {
        m_chiSquaredVal = currentChiVal;
        splitIndex = next - 1;
     }
```

```
      [...]

   }

   // Was there any useful split?
   if ((m_index == 0) || (splitIndex < 0)) {
      return;
   }

   [...]

   double originalSig = chiSquareCrit.splitCritValue(m_distribution);
   // Compute Bonferroni correction for best split.
   m_chiSquaredProb = originalSig * chiSquareCrit.bonferroniFactor(
      m_index, m_distribution.numBags(), hasMissings, true);
   if (Utils.gr(m_chiSquaredProb, m_SigLevel)) {
      // Not significant: merge all values
      m_distribution = new Distribution(mergeAll(m_distribution.matrix()));
   }
}
```

Destacan tres diferencias respecto al método original: la primera que aquí sí se tienen en cuenta los *missings*; la segunda, que se cambia el valor de la entropía por el estadístico χ^2 como criterio para evaluar cada posible punto de división; por último, que aplica la corrección de Bonferroni al valor de probabilidad de χ^2 asociado a la división generada.

6.3.1.4. ChiSquareSplitCrit

Hereda de la clase abstracta SplitCriterion, y sirve para representar el criterio de división de un nodo para este algoritmo. Mientras que en J48 estos criterios estaban asociados a cálculos relacionados con la entropía, en este caso el criterio está ligado al cálculo de χ^2.

El único método que se sobreescribe en esta clase es el siguiente:

splitCritValue()

Calcula la probabilidad del test de la χ^2 para una tabla de contingencia -o distribución, como se le llama en el código fuente de Weka- dada, que es el criterio de división utilizado por CHAID y CHAID*.

```
public double splitCritValue(Distribution bags) {
   return ContingencyTables.chiSquared(bags.matrix(), false);
}
```

Los siguientes, en cambio, son métodos nuevos añadidos aquí para facilitar cálculos comunes relacionados con el proceso de buscar la mejor división para un nodo:

chiVal()

Calcula el estadístico χ^2 para una tabla de contingencia dada.

```
public double chiVal(Distribution bags) {
   return ContingencyTables.chiVal(bags.matrix(), false);
}
```

bonferroniFactor()

Calcula el factor de Bonferroni para atributos de tipo nominal y ordinal, teniendo en cuenta si existen valores *missing* o no.

```
public double bonferroniFactor(int c, int r, boolean hasMissing, boolean ordered) {

   double b;
   if (hasMissing) { // One element more
      c++;
   }

   if (ordered) {
      // Discrete or Enumerated but Ordinal attributes
      if (hasMissing) {
         b = combinations(c-2, r-2) + r * combinations(c-2, r-1);
      } else {
         b = combinations(c-1, r-1);
      }
   } else {
      // Discrete or Enumerated but Nominal attributes
      b = BFfactor(c, r);
   }

   return b;
}
```

7. EXPERIMENTACIÓN

7.1. Preparando el experimento

7.1.1. Bases de datos utilizadas

Existen varios sitios donde se pueden conseguir bases de datos compatibles con Weka. Sin ir más lejos, la propia página web de Weka [Hal-09b] cuenta con una sección *Datasets* donde ofrece distintas colecciones de bases de datos para diferentes problemas de aprendizaje automático como clasificación, regresión, etc.

Pero el sitio más conocido y con mayor variedad de bases de datos es el repositorio de la UCI [Lich-13]. De hecho, algunas de las colecciones que ofrece la sección *Datasets* de Weka están extraídas de este repositorio.

El grupo ALDAPA, a lo largo de las distintas publicaciones que ha realizado en relación a algoritmos de clasificación, ha ido elaborando una colección de bases de datos obtenidas en dicho repositorio con las que ha realizado sus diferentes experimentaciones.

En el momento de su publicación más reciente [Iba-15], esta colección cuenta con 36 bases de datos que cubren toda la variedad de casos necesarios para probar el algoritmo presentado, BFPART.

Para la fase de experimentación de este proyecto se ha utilizado esta misma colección, puesto que el abanico de casos que interesa probar en CHAID* es el mismo que para BFPART: variables nominales, ordinales y continuas, y presencia de valores *missing*.

En la tabla 7.3 se muestra el listado de bases de datos incluidas, indicando sus características.

		Nº atributos						
Características		Discretas						
	Nº casos	Nominales	Ordinales	Contínuas	**Total**	**Nº clases**	**Dist. Orig**	**Valores Missing**
breast-w	699		10		10	2	34,5	y
heartc	303	7	1	5	13	2	45,87	y
spam	4601			57	57	2	39,4	n
hypo	3163	18		7	25	2	4,77	y
liver	345			6	6	2	42,03	n
lymph	148	17	1		18	4	1,35	n
lymph2	148	17	1		18	2	41,22	n
credit_a	690	8		6	14	2	44,49	n
vehicle	846			18	18	4	23,52	n
vehicle2	846			18	18	2	23,52	n
iris	150			4	4	3	33,33	n
iris2	150			4	4	2	33,33	n
glass	214			9	9	7	4,2	n
glass2	214			9	9	2	23,83	n
breast-y	286	5	4		9	2	29,72	y
voting	435	16			16	2	38,62	y
heart-h	294	8		5	13	2	36,05	y
hepatitis	155	13		6	19	2	20,65	y
credit_g	1000	10	3	7	20	2	30	n
soybean_15CL	290	34	1		35	15	3,45	y
soybean_15CL2	290	34	1		35	2	13,79	y
segment2310	2310			19	19	7	14,29	n
segment2310_2	2310			19	19	2	14,29	n
segment210	210			19	19	7	14,29	n
segment210_2	210			19	19	2	14,29	n
sick-euthyroid	3164	18		7	25	2	9,26	n
bands	540	18		21	39	2	42.2	y
ks-vs-kp	3196	36			36	2	47.8	n
optdigits	5620		64		64	2 <= 10	9,9	n
car	1728		6		6	2 <= 4	30.0	n
abalone	4177			8	8	2 <= 29	8,6	n
solar_flare	1389	13			13	2	15,7	n
yeast	1484			8	8	2 <= 10	28,9	n
splice_junction	3190	60			60	2 <= 3	24,1	n
kddcup	4941	7		34	41	2	19,69	n
pima	768			8	8	2	34,9	n

Tabla 7.1: Colección de 36 problemas de clasificación.

7.1.2. Criterios evaluados

De entre todos los criterios que ofrece el *Experimenter* de Weka para medir la bondad de los clasificadores, aquí nos hemos centrado en los resultados para cinco en particular:

- Tasa de acierto: en Weka se muestra seleccionando la opción *Percent_correct*, y mostrará el porcentaje de casos clasificados correctamente, por lo que cuanto mayor sea el valor obtenido, mejor se considera el resultado.

- AUC (*Area Under ROC Curve*): es una de las medidas más utilizadas para evaluar la bondad de un clasificador, y se considera una de las más robustas. En Weka se muestra seleccionando la opción *Weighted_avg_area_under_ROC*. En este caso el resultado también es mejor cuanto mayor sea el valor obtenido.

- Tiempo de construcción del clasificador: aunque este criterio no habla sobre la capacidad de clasificar correctamente, también es interesante comparar el tiempo de cómputo que necesita cada clasificador para la fase de *entrenamiento* o de construcción del modelo. En Weka se corresponde con la opción *UserCPU_Time_training*.

- Número de nodos: este recuento incluye nodos internos y hojas. En Weka se muestra mediante la opción *measureTreeSize*. Este criterio tampoco habla sobre la capacidad de clasificar los casos correctamente, pero sí ofrece una medida de la complejidad del modelo construído, o dicho de otra manera: su capacidad explicativa. A misma tasa de acierto, siempre es preferible que el modelo sea lo menos complejo posible, así que cuanto menor sea el valor de este criterio, mejor.

- Número de nodos hoja del árbol. En Weka es la opción *measureNumLeaves*. Al igual que el anterior criterio, es otra manera de medir la complejidad del árbol.

7.1.3. Configuración del experimento

En el apartado 5.3.2 se explicó cómo configurar un experimento mediante la pestaña *Setup* del experimentador de Weka. Para este experimento en particular, se han utilizado las siguientes opciones, que también coinciden con las utilizadas para la experimentación con el algoritmo BFPART en [Iba-15]:

En la sección *Experiment Type* se ha seleccionado el tipo de validación "*Cross-validation*" con "*Number of folds*" de 10, que es su valor por defecto.

En la sección *Datasets* se han añadido las 36 bases de datos mencionadas en el apartado 7.1.1, de las cuales 10 se han pasado previamente a formato XRFF -explicado en el apartado 5.4.2- para poder marcar las variables de tipo ordinal como tales.

En la sección *Iteration Control* se ha elegido hacer 5 repeticiones de cada test realizado para cada algoritmo y conjunto de datos.

En la sección *Algorithms*, además de JCHAIDStar, se ha añadido el J48 de cara a evaluarlo también con los mismos criterios de bondad y poder comparar los resultados de ambos algoritmos. J48 es el nombre de la implementación en Weka del algoritmo clásico C4.5, explicado en el apartado 4.2.

7.2. Resultados

Los resultados analizados a continuación se han extraído de la pestaña *Analyse* del experimentador de Weka, ejecutando un test para cada criterio de bondad explicado en el apartado anterior.

7.2.1. Tasa de acierto

Para este primer criterio, en la tabla 7.2 puede observarse que los resultados obtenidos por C4.5 son, en promedio, algo mejores que los de CHAID*: una tasa del 87.31% de aciertos del primero frente a un 85.83% del segundo.

Sin embargo, atendiendo al detalle de los distintos casos testeados se observa que solo 6 de los resultados favorables al C4.5 son estadísticamente significativos, frente a 1 caso a favor de CHAID*. Para los 29 casos restantes, no se considera que la diferencia sea significativa.

Percent_correct		
Dataset	**JCHAIDStar**	**J48**
abalone2	91.33	91.33
bands	72.78	76.37
Glass	64.89	69.05
Glass2	92.13	93.06
heart-h	81.31	80.57
hepatitis	77.23	79.74
hypo	99.01	99.27
iris	95.73	94.93
iris2	100.00	99.33
kddcup	99.21	99.54 v
kr-vs-kp	99.18	99.44
liver	66.69	66.38
pima	74.32	74.85
segment210	84.19	88.19
segment210_2	97.33	97.90
segment2310	94.07	96.76 v
segment2310_2	99.37	99.32
sick_euthyroid	95.71	98.00
solar_flare	84.16	84.28
spambase	90.25	92.76 v
splice_junction2	94.26	95.98 v
vehicle	68.79	71.87
vehicle2	93.47	94.04
voting	94.52	96.55 v
yeast2	75.52	75.54
breast-w_XRFF	91.93	94.45
breast-y_XRFF	75.39	74.33
car2_XRFF	97.41	94.83 *
credit_g_XRFF	69.92	71.30
heartc_XRFF	78.48	74.71
lymph_XRFF	73.36	78.23
lymph2_XRFF	80.01	77.08
optdigits2_XRFF	93.49	93.87
soybean15CL_XRFF	65.93	89.38 v
soybean2_XRFF	93.66	94.21
credit_a	84.90	85.88
Average	**85.83**	**87.31**
	(v/ /*)	(6/29/1)

Tabla 7.2: Resultados para la tasa de acierto.

7.2.2. AUC

Para este segundo criterio, en cambio, según los resultados mostrados en la tabla 7.3 CHAID* es algo mejor que C4.5 en promedio, si bien hay una mejora estadísticamente significativa a favor de C4.5 para 4 de los casos testeados frente a un solo valor estadísticamente significativo a favor de CHAID.

Weighted_avg_area_under_ROC		
Dataset	**JCHAIDStar**	**J48**
abalone2	0.5000	0.5000
bands	0.7620	0.7790
Glass	0.7907	0.8141
Glass2	0.8962	0.8767
heart-h	0.7963	0.7809
hepatitis	0.6336	0.6720
hypo	0.9577	0.9530
iris	0.9719	0.9653
iris2	1.0000	0.9900
kddcup	0.9897	0.9937
kr-vs-kp	0.9956	0.9981 v
liver	0.6414	0.6499
pima	0.7843	0.7571
segment210	0.9452	0.9378
segment210_2	0.9661	0.9461
segment2310	0.9846	0.9865
segment2310_2	0.9855	0.9838
sick_euthyroid	0.9287	0.9462
solar_flare	0.5220	0.5870 v
spambase	0.9239	0.9383
splice_junction2	0.9646	0.9604
vehicle	0.8605	0.8506
vehicle2	0.9549	0.9392
voting	0.9479	0.9778 v
yeast2	0.7138	0.7275
breast-w_XRFF	0.9391	0.9645
breast-y_XRFF	0.6649	0.6038
car2_XRFF	0.9959	0.9799 *
credit_g_XRFF	0.6547	0.6441
heartc_XRFF	0.8081	0.7608
lymph_XRFF	0.7790	0.8090
lymph2_XRFF	0.8134	0.7677
optdigits2_XRFF	0.9003	0.8834
soybean15CL_XRFF	0.9418	0.9661 v
soybean2_XRFF	0.8404	0.8236
credit_a	0.8674	0.8826
Average	**0.8506**	**0.8499**
	(v/ /*)	(4/31/1)

Tabla 7.3: Resultados para el AUC.

7.2.3. Tiempo de construcción del clasificador

Los resultados de la tabla 7.4 indican que un árbol CHAID* es 2,33 veces más costoso de construir que un C4.5 en promedio.

UserCPU_Time_training		
Dataset	**JCHAIDStar**	**J48**
abalone2	250	214
bands	246	0.0096 *
Glass	28	28
Glass2	20	14
heart-h	12	20
hepatitis	10	16
hypo	294	0.0180 *
iris	0	0
iris2	0	0
kddcup	1.140	0.0474 *
kr-vs-kp	184	134
liver	8	18
pima	46	44
segment210	50	34
segment210_2	14	8
segment2310	744	0.0320 *
segment2310_2	260	220
sick_euthyroid	1.790	0.0372 *
solar_flare	18	16
spambase	3.604	0.3092 *
splice_junction2	456	0.0158 *
vehicle	164	126
vehicle2	96	64
voting	54	10
yeast2	84	112
breast-w_XRFF	188	0.0006 *
breast-y_XRFF	10	6
car2_XRFF	88	0.0012 *
credit_g_XRFF	90	98
heartc_XRFF	26	22
lymph_XRFF	16	2
lymph2_XRFF	16	10
optdigits2_XRFF	3.878	0.0440 *
soybean15CL_XRFF	1.094	0.0020 *
soybean2_XRFF	14	12
credit_a	42	32
Average	**418**	**179**
	(v/ /*)	(0/25/11)

Tabla 7.4: Resultados para el tiempo de construcción del árbol.

7.2.4. Complejidad del árbol

Como puede observarse en la tabla 7.5, JCHAIDStar consigue construir árboles que, en promedio, tienen la mitad de nodos que los árboles construidos por J48. Los resultados muestran una diferencia estadísticamente significativa a favor de JCHAIDStar en 25 de las 36 bases de datos utilizadas.

Desde el punto de vista de la capacidad explicativa, este es un muy buen resultado ya que, como se ha dicho, cuanto menos complejo es el árbol, se considera más explicativo.

measureTreeSize		
Dataset	**JCHAIDStar**	**J48**
abalone2	1.00	1.00
bands	47.70	94.00 v
Glass	14.68	45.72 v
Glass2	14.40	12.60
heart-h	5.24	9.58 v
hepatitis	9.22	17.72 v
hypo	8.12	12.28 v
iris	8.36	8.28
iris2	3.00	3.00
kddcup	37.20	105.66 v
kr-vs-kp	64.04	55.68 *
liver	12.04	49.32 v
pima	19.64	41.24 v
segment210	21.72	24.80 v
segment210_2	6.96	7.88
segment2310	81.28	80.96
segment2310_2	15.36	18.96 v
sick_euthyroid	29.78	25.24
solar_flare	2.68	4.32 v
spambase	235.32	206.96 *
splice_junction2	40.74	139.56 v
vehicle	82.32	137.28 v
vehicle2	38.40	40.44
voting	4.38	10.68 v
yeast2	18.44	66.32 v
breast-w_XRFF	15.10	32.00 v
breast-y_XRFF	05.08	11.90
car2_XRFF	53.92	94.48 v
credit_g_XRFF	29.42	126.74 v
heartc_XRFF	16.08	41.98 v
lymph_XRFF	8.16	29.00 v
lymph2_XRFF	10.14	25.28 v
optdigits2_XRFF	60.42	435.52 v
soybean15CL_XRFF	23.72	71.90 v
soybean2_XRFF	5.98	11.26 v
credit_a	7.56	32.98 v
Average	**29.38**	**59.24**
	(v/ /*)	(25/9/2)

Tabla 7.5: Resultados para el número de nodos totales.

Desde el punto de vista del número de nodos hoja, en la tabla 7.6 puede observarse que los resultados de JCHAID* respecto a J48 son incluso mejores que para el anterior criterio: una media de 16 nodos hoja frente a 41.

measureNumLeaves		
Dataset	**JCHAIDStar**	**J48**
abalone2	1.00	1.00
bands	25.38	58.08 v
Glass	7.84	23.36 v
Glass2	7.70	6.80
heart-h	3.12	5.76 v
hepatitis	5.14	9.36 v
hypo	4.56	6.64 v
iris	4.68	4.64
iris2	2.00	2.00
kddcup	19.10	92.60 v
kr-vs-kp	32.52	29.34 *
liver	6.52	25.16 v
pima	10.32	21.12 v
segment210	11.36	12.90 v
segment210_2	3.98	4.44
segment2310	41.14	40.98
segment2310_2	8.18	9.98 v
sick_euthyroid	15.74	13.12
solar_flare	1.84	2.72
spambase	118.16	103.98 *
splice_junction2	22.04	122.24 v
vehicle	41.66	69.14 v
vehicle2	19.70	20.72
voting	2.70	5.84 v
yeast2	9.72	33.66 v
breast-w_XRFF	10.38	28.90 v
breast-y_XRFF	03.04	8.94
car2_XRFF	30.68	69.06 v
credit_g_XRFF	15.92	90.22 v
heartc_XRFF	8.74	27.24 v
lymph_XRFF	5.34	20.06 v
lymph2_XRFF	06.08	16.86 v
optdigits2_XRFF	43.32	409.96 v
soybean15CL_XRFF	13.64	49.24 v
soybean2_XRFF	3.98	8.38 v
credit_a	4.32	22.26 v
Average	**15.88**	**41.02**
	(v/ /*)	(24/10/2)

Tabla 7.6: Resultados para el número de nodos hoja.

7.3. Análisis de los resultados

Las diferencias encontradas entre los resultados obtenidos por este algoritmo y los de C4.5 eran fácilmente predecibles. Por ejemplo, la reducción tan significativa de la complejidad de los modelos construidos por CHAID* es una consecuencia lógica de la fase de *Merging* -o *unión de categorías*- que realiza para las variables nominales y ordinales: mientras que C4.5 divide estos nodos en tantas ramas como categorías existan para el atributo, CHAID* busca combinarlas de la manera más óptima en grupos de categorías.

Por otro lado, esta búsqueda de combinaciones también explica la caída de rendimiento en cuanto al tiempo de construcción del clasificador: a mayor número de categorías, mayor cantidad de combinaciones a explorar, aumentando drásticamente el coste computacional.

Finalmente, se ha observado que los resultados de la validación de los modelos generados son similares a los obtenidos por otras implementaciones existentes, como la versión del grupo ALDAPA integrada en la plataforma Haritza. En dicha versión, también se observaba que C4.5 obtenía una tasa de acierto ligeramente superior a la de CHAID*, mientras que éste quedaba por encima en cuanto al valor del *Area Under ROC Curve*. El mayor coste computacional y la menor complejidad de los árboles generados también son efectos que se habían observado en aquella experimentación.

8. CONCLUSIONES

En este proyecto hemos partido de un algoritmo clásico del aprendizaje automático como es el CHAID, y de una aplicación de software libre de Data Mining como es Weka donde faltaba una implementación de ese algoritmo.

Tras aprender los fundamentos básicos para entender el algoritmo y haber analizado la arquitectura de software de Weka, hemos conseguido añadir una implementación de CHAID en la aplicación bajo el nombre de JCHAID.

También se ha añadido una extensión de este clasificador bajo el nombre JCHAIDStar que implementa el CHAID*, una adaptación diseñada por el grupo ALDAPA para añadir al algoritmo original la posibilidad de trabajar con variables continuas.

El resultado de este desarrollo se ha validado con un trabajo de experimentación que creemos que puede considerarse robusto, tanto por el volumen de datos utilizado para los tests -36 bases de datos con problemas de clasificación de características muy diversas- como por la variedad de criterios utilizados para analizar los resultados del experimento. De hecho, son las mismas bases de datos y criterios utilizados en [Iba-15], artículo recientemente aceptado para su publicación, lo que respalda la afirmación de que la experimentación aportada aquí es robusta.

Estos resultados muestran que la implementación realizada es válida. De todas maneras, no la consideraría una solución cerrada y definitiva, ya que aún quedarían tareas a realizar que comentaremos a continuación como líneas abiertas del proyecto.

Aun con estas líneas abiertas en mente, considero que hemos alcanzado con éxito las expectativas iniciales del proyecto, obteniendo dos clasificadores que funcionan según lo esperado.

En un plano más personal, cuando recibí esta propuesta de proyecto me atrajo la posibilidad de tener una aproximación más práctica al mundo del aprendizaje automático y la minería de datos. Aunque las asignaturas relacionadas con la Inteligencia Artificial me habían parecido interesantes durante la carrera, mi sensación era la de tener una visión muy teórica y abstracta del tema.

Este proyecto me ha servido para conocer aplicaciones reales de estas ramas de la Informática que a veces pueden parecernos limitadas al ámbito académico y de investigación, pero que también sirven y se utilizan para resolver problemas más cotidianos.

Además, la posibilidad de que el proyecto pueda tener continuidad dentro de alguna de las líneas de trabajo del grupo ALDAPA o incluso ser compartido con la comunidad de desarrolladores de Weka añade la satisfacción de estar entregando un trabajo que tiene una verdadera utilidad.

8.1. Líneas abiertas

Algunos de los siguientes puntos son tareas que simplemente quedaban fuera del alcance del proyecto, mientras que otros corresponden a posibles mejoras detectadas en las fases finales del desarrollo, donde no hubo tiempo para abordarlos:

- Implementar la fase de división de grupos de categorías, que como se explicaba en el apartado 4.3.1, siendo una fase opcional del algoritmo se decidió dejar fuera del alcance del proyecto.

- Según se explica en [Per-99], el algoritmo CHAID puede recibir 4 parámetros de los cuales aquí solo se ha implementado uno, el *nivel de significancia*. Los demás parámetros, por herencia de J48, tienen sus valores fijados estáticamente en el código. Habría que parametrizar esos 3 valores pendientes para que el usuario los pueda modificar a través de la interfaz de usuario de Weka.

- Exportar JCHAID y JCHAIDStar como paquetes de Weka:
  Desde las últimas ramas de versiones de Weka (3.8.x para la versión estable y 3.9.x para la versión de desarrollo), existe la posibilidad de desarrollar nuevos componentes (clasificadores, filtros, etc) a modo de paquetes. En estas ramas, Weka incluye un componente llamado *Package Manager* que permite instalar dichos paquetes, añadiendo a la aplicación los componentes que éstos incluyan. Es una forma más limpia y sencilla de distribuir componentes como podrían ser los clasificadores JCHAID y JCHAIDStar.

- Implementar la consolidación de CHAID: CTCHAID.
  Recientemente, ALDAPA propuso en [Iba-15b] otra extensión de CHAID llamada CTCHAID, basándose en la técnica de Construcción de Árboles Consolidados que ya se había aplicado anteriormente al algoritmo C4.5. Una de las motivaciones que se explicaban en el apartado 1.1 para este proyecto era la de integrar en Weka una implementación del CHAID clásico que sirviera como base para implementar extensiones del mismo, como este CTCHAID o el CHAID*.
  Ya que el CHAID* también se ha abordado en este proyecto, el siguiente paso sería integrar el algoritmo CTCHAID.

- Copiar el tratamiento de variables nominales de J48 en CHAID, permitiendo desactivar mediante algún parámetro el algoritmo de Unión/División de categorías explicado en el apartado 4.3. De esta manera, podrían verse los resultados de J48 y JCHAID cuando aplican el mismo tratamiento a las variables nominales, pudiendo comparar el efecto del aplicar el criterio de división de χ^2 y el de la entropía, para comprobar cuál de los dos criterios es mejor en igualdad de condiciones.

- Generalizar la función *mergeValues()* de CHAIDDistribution de manera que sea capaz de trabajar con distintos criterios de división como la entropía, en lugar de funcionar solo con el criterio de χ^2. De esta manera podría aplicarse el algoritmo de Unión/División de categorías también al J48. Esto sería interesante porque en la experimentación de este proyecto, J48 ya ha dado mejores resultados que JCHAIDStar en criterios como la tasa de acierto, pero quizá se pueda mejorar aun más refinando su tratamiento de variables nominales o, al menos, reducirse la complejidad de los árboles que genera para este tipo de variables.

9. BIBLIOGRAFÍA

9.1. Publicaciones

- [Arb-13] Olatz Arbelaitz, O, Ibai Gurrutxaga, Fernando Lozano, Javier Muguerza, Jesús M. Pérez. “J48Consolidated: An implementation of CTC algorithm for WEKA”, (2013) Technical Report EHU-KAT-IK-05-13, University of the Basque Country (UPV/EHU)
- [Qui-93] Quinlan, J.R. “C4.5: Programs for Machine Learning”. Morgan Kaufmann Publishers, 1993.
- [Qui-86] Quinlan, J.R. “Induction of Decision Trees”. Machine Learning, 1, págs. 81-106, 1986
- [Iba-15] Igor Ibarguren, Aritz Lasarguren, Jesús M. Pérez, Javier Muguerza, Olatz Arbelaitz, Ibai Gurrutxaga. “BFPART: Best-first PART”. Submitted to Information Sciences (2015).
- [Iba-15b] Igor Ibarguren, Jesús M. Pérez, Javier Muguerza. “CTCHAID: extending the application of the consolidation methodology”, (2015) Progress in Artificial Intelligence.
- [Kas-80] Kass, G.V. “An Exploratory Technique for Investigating Large Quantities of Categorical Data”. Appl. Statist., Vol. 29, Nº 2, págs. 119-127, 1980 (http://www.jstor.org/stable/2986296)
- [Per-99] Pérez, J.M. “Un sistema de ayuda a la detección del fraude en compañías de seguros”, Dpto. de Arquitectura y Tecnología de Computadores, 1999.
- [Lich-13] Lichman, M. (2013). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.
- [Bou-16] Remco R. Bouckaert, Eibe Frank, Mark Hall, Richard Kirkby, Peter Reutemann, Alex Seewald, David Scuse. “Writing a new Classifier”

(2016) WEKA Manual for Version 3-6-14, págs 229-235. University of Waikato.

9.2. Sitios web

- [Mug-16] Muguerza, Javier. “Aldapa”, (2016) Aldapa, Algorithms, Data Mining and Parallelism [http://www.aldapa.eus/es/aldapa/co-13/]. UPV/EHU, Dpto. de Arquitectura y Tecnología de Computadores.
- [Hal-09a] Mark Hall, M, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, Ian H. Witten. “Weka 3: Data Mining Software in Java”, (2009) Machine Learning Group [http://www.cs.waikato.ac.nz/ml/weka/index.html]. University of Waikato.
- [Hal-09b] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, Ian H. Witten. “Collections of Datasets”, (2009) Machine Learning Group [http://www.cs.waikato.ac.nz/ml/weka/datasets.html]. University of Waikato.
- [Smi-07] Smith, B. “A Quick Guide to GPLv3.”, (2007). GNU Operating System [http://www.gnu.org/licenses/quick-guide-gplv3.html]. Free Software Fundation.
- [Ibm-12] IBM, “Merging (CHAID Algorithms)”, (2012). CHAID Algorithm [http://www.ibm.com/support/knowledgecenter/SSLVMB_21.0.0/com.ibm.spss.statistics.help/alg_tree-chaid_algorithm_merging.htm]. IBM Knowledge Center.