eman ta zabal zazu

Universidad del País Vasco Euskal Herriko Unibertsitatea

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

Machine detection of emotions: Feature Selection

Final Degree Dissertation
Degree in Mathematics

Leire Santos Moreno

Supervisor:
Raquel Justo Blanco
María Inés Torres Barañano

Leioa, 31 August 2016

Contents

Introduction

The aim of this project is to study the capacity of machines to detect emotions in written text. Humans are experts at perceiving emotions in oral language and, although it can be more challenging, even in a written context our ability to distinguish emotions is astonishing. Nowadays, the amount of written text generated every second is huge. Thus, achieving that emotion recognition by machines is a topic in which a lot of research has been done lately, as it becomes impossible to analyze the emotions manually for very large databases. Therefore, the present is a challenging but very interesting and useful issue.

Sarcasm is one of the human emotions that is most difficult to notice, it can be tough to recognize even for a person. In this work sarcasm recognition done by machine will be studied, by using statistical pattern classification techniques to try to classify sentences in Spanish as sarcastic or not sarcastic. This will be done with a special focus on the part of selecting the features to carry out the classification, by studying and experimenting with different dimensionality reduction techniques.

This document is divided into three chapters. In Chapter 1 the main idea of how a pattern classification is done is explained, starting with the basic definitions and doing a brief review of the steps needed to achieve the classification. A concrete type of statistical classifier is also presented, the Naive Bayes classifier.

Chapter 2 is focussed on the issue of dimensionality reduction. The concept of dimensionality reduction is explained, as well as the benefits that it provides. Two different families of dimensionality reduction methods are presented, feature selection and feature transformation, and examples of methods in both families are given.

Finally, Chapter 3 presents the experiments done. With these experiments, theoretical knowledge is reinforced while dealing with a real problem. Different dimensionality reduction methods are tested in data taken from social networks and the results are discussed, in order to determine which method works better in this precise context.

Chapter 1

Introduction to pattern classification

In this chapter the basic idea of pattern classification is explained. Different steps of the classification process are analyzed and one type of classifier, Naive Bayes classifier, is presented.

1.1 Classification process

Classification process is divided into different steps. First of all, features to be used in the classification need to be extracted. Then, the classifier is trained and how it works is evaluated. All that process is recursive, information on how the classifier is working can be used to choose more appropriate features or to train the classifier differently.

1.1.1 Features and patterns

Definition 1.1.1. (Feature) Let O be an object that can be classified in one of the following C classes $\{\omega_1, \omega_2, \dots, \omega_C\}$. A *feature*, X, is a random variable that represents one property of that object and which will help classifying the object O in one of the possible classes.

When more than one feature can be measured in one object, a *feature vector* $\mathbf{X}$ is constructed, $\mathbf{X} = (X_1, X_2, \dots, X_m)$ where each X_i is a feature of the object O for $i \in \{1, 2, \dots m\}$, with $m \in \mathbb{N}$.

Definition 1.1.2. (Pattern) Given O an object that can be classified in C different classes $\{\omega_1, \omega_2, \dots, \omega_C\}$ and $\mathbf{X} = (X_1, X_2, \dots, X_m)$ a feature vector for that object, a *pattern* is a m-dimensional data vector $\mathbf{x} = (x_1, x_2, \dots, x_m)$ where each of its components x_i is a measurement of one feature X_i [3, Chapter 1]. For each pattern a categorical variable z is defined, the *label*, such that if $z = i$, the object that gave this pattern belongs to class ω_i.

Example 1.1.1. (Based on the example used in [1]) Suppose that the object O is a fish, that can be classified into two classes: $\omega_1 =$ salmon or $\omega_2 =$ sea bass. Then, the features that can be measured in the fish are the length of the animal, its width, its weight, the brightness of the scales, the diameter of the eye, ... Thus, one possible feature vector in this case could be $\mathbf{X} = (X_1, X_2)$ where X_1 is the length of the fish (measured in meters) and X_2 the weight (in kilograms).

Now, given a concrete fish, values for the features in $\mathbf{X}$ can be measured for that precise individual and obtain a pattern, for example $\mathbf{x} = (0.5, 6)$. Moreover, if it is known that the fish with that pattern is a salmon, its label will be $z = 1$, as it belongs to class ω_1.

Remark 1.1.1. In order to classify sentences into sarcastic or not sarcastic, the features that are going to be used are the counts of words and groups of words, i.e, how many times each word or each sequence of words appears in that sentence. This is further developed in Chapter 3.

The process of obtaining a pattern $\mathbf{x}$ from an object O is called *feature extraction*. It involves measuring the values for the features that are going to be used in the classification process and this can be done in a wide variety of ways, which depend on the type of feature measured. For example, in the case of Example 1.1.1, the values for the length of an individual can be obtained by photographing the fish and sending the picture to a computer that uses the it to calculate the length of the animal; while the weight can be obtained from a digital scale. During this feature extraction process some type of preprocessing of the data can be done, to help simplifying the following operations. Moreover, in some cases dimensionality reduction is done at this step, as it will be explained in Chapter 2.

1.1.2 Training the classifier

The aim of patter recognition is to, given a pattern $\mathbf{x}$ for which the label z is unknown, be able to decide which is the value of z. That is, to classify the object which generated that pattern in one of the possible classes. The classifier can be represented as a function Φ that assigns a class from the set of all the possible classes, Ω, to each object O from the set of all the objects $\mathcal{O}$.

$$\begin{array}{rccc} \Phi & : & \mathcal{O} & \rightarrow & \Omega \\ & & O & \mapsto & \Phi(O) = \omega_i \end{array}$$

The exact form of the function Φ is determined during the process of training the classifier.

Definition 1.1.3. (Training set) The *training set* is a large set of objects from which a set of patterns $\{\mathbf{x}_1, \mathbf{x}_1, \ldots, \mathbf{x}_N\}$ is obtained. That training set will be used to design the classifier, that is, to tune the parameters that the classifier uses.

Remark 1.1.2. The training set can be provided with the corresponding class for each object. In this way the set that is used for training the classifier is a set of patterns together with their labels $\{(\mathbf{x}_1, z_1), (\mathbf{x}_1, z_2), \ldots, (\mathbf{x}_N, z_N)\}$. This learning process, that involves using the known labels, is called *supervised learning.*

On the other hand, when the labels are not provided, the learning is called *unsupervised learning.* There exists also an option in between the previous two, *reinforcement learning*, in which the only information provided while learning is if the label $\hat{z}$ that the classifier has assigned to a pattern $\mathbf{x}$ is the correct one or not, but the correct label is not provided. Notice that in the case of having only two categories reinforcement learning is the same as supervised learning.

Regarding the construction of the classifier, there exist different classification techniques that can be applied. *Statistical* pattern recognition focuses on statistical properties of the data, this is for example the case of the Naive Bayes [Section 1.2]. *Neural network* pattern classification is a close descendant of the statistical approach but in this case the decision is based on the response of a network of processing units to an input stimuli. Finally, another classification method is the *syntactic* pattern recognition, where rules or grammars decribe the decision [1] [2].

1.1.3 Evaluation measures

Once the classifier is built, it is necessary to have some way to measure how well it works.

Definition 1.1.4. (Test set) The *test set* is a set of objects for which the classes are already known. When a pattern $\mathbf{x}$ corresponding to the an object on the test set is passed to the classifier, the classifier evaluates it and returns a predicted label $\hat{z}$ for that pattern. As the real label z is known, z and $\hat{z}$ can be compared in order to evaluate the performance of the classifier.

One of the most common ways to evaluate the performance of a classifier is calculating its error rate, that is, the proportion of samples incorrectly classified. However, many other metrics can be calculated from the confusion matrix. Table 1.1 shows a 2×2 confusion matrix for the two-category case in which only two possible categories exist, the positive and the negative. Table 1.2 shows some of the most common evaluation metrics to assess the performance of the classifier in the two-category case. Many of those metrics are also extendable to the multiclass problems [3].

		True label (z)	
		Positive	Negative
Predicted label ($\hat{z}$)	Positive	True positive (tp)	False positive (fp)
	Negative	False negative (fn)	True negative (tn)

Table 1.1: 2×2 confusion matrix

Name	Definition
Accuracy	$\frac{tp+fp}{tp+tn+fp+fn}$
Error rate	1- Accuracy
False positive rate	$\frac{fp}{fp+tn}$
Recall (or true positive rate)	$\frac{tp}{tp+fn}$
Precision	$\frac{tp}{tp+fp}$
F1	$\frac{2(\text{precision}\times\text{recall})}{(\text{precision}+\text{recall})}$

Table 1.2: Performance assessment metrics for the two-category case.

Remark 1.1.3. The score that will be used as evaluation measure for the experiments in this work will be the F1 score. This is the harmonic mean of precision and recall, which represent respectively the proportion of true positives in all the predicted positives and the proportion of real positives that the classifier correctly predicted as positive. It is important, while using the F1 score, to also check precision and recall, since it is possible to obtain a high F1 value even if one of those has a low value. Imagine for example, that half of the labels should be positive and half negative, but the classifier used states that all of them are positive. Then, the value for precision would be 1 and the value for recall 0.5, meaning that even if the prediction would not be correct at all, the F1 score would be 0.667, which apparently is not so bad.

1.1.4 K-fold Cross Validation

As it has been seen, during the process of classification data examples are needed in two steps: to train the classifier by choosing the right parameters and to measure how well the classifier performs. Thus, a large number of data is needed to conform the training and the test set. Due to the cost and difficulty of obtaining that data, a process called *Cross Validation* is sometimes carried out.

K-fold Cross Validation consists in dividing the available data into K parts. Then, K experiments will be done and in each of the experiments

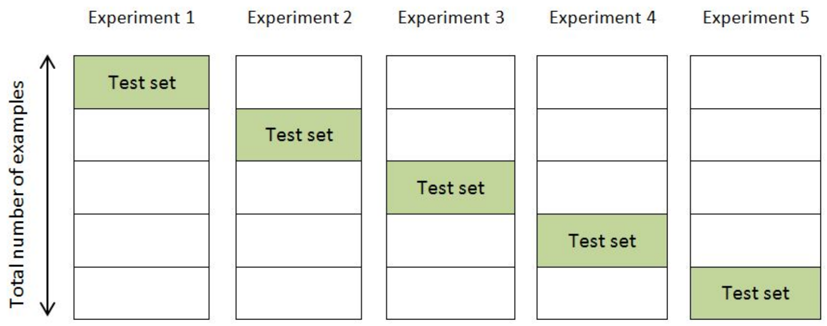

Figure 1.1: Scheme of Cross Validation process for $k = 5$. Total data is divided into 5 parts and five experiments are performed. In each experiment the green part of the data is used as test set while the rest is used as training set.

one of the K folds will be the test set while the remaining $K - 1$ folds conform the training set. In this way, all the data is used both for training and for testing. The true value of an evaluation measure will be calculated as the average of the value obtained in each experiment. Figure 1.1 shows an example of the scheme of the Cross Validation process.

The choice of K is done depending on the size of the dataset, taking into account that a large number of folds gives more accurate results but needs more computation time, while a smaller K will result into faster but less accurate outcome. A common choice for K is $K = 10$ [2]. During the performance of the experiments in this work, a 10-fold Cross Validation is applied (see Chapter 3).

1.2 Naive Bayes classifier

Naive Bayes classifier is a type of statistical classifier based on Baye's theorem and the naive assumption that the variables X_i are independent.

Definition 1.2.1. (State of nature) Given an object O, ω will denote the *state of nature* of O, with $\omega = \omega_i$ if the element belongs to the category ω_i. That is, the state of nature is the real class to which the object belongs.

Let $P(\omega_i)$ denote the prior probability, i.e., how likely it is to have an element with state of nature ω_i. The prior probability can be used to create a decision rule: if $P(\omega_i) > P(\omega_j) \forall j \neq i$, decide ω_i. However, this rule makes sense only for the classification of one object. If more than one element need to be classified then all of them would be assigned to the category with higher prior probability. Thus, more information is needed in order to create a more complex decision rule that will accurately classify the data.

Let $\mathbf{X}$ be a feature vector,a random vector whose distribution depends on the state of nature, then $p\left(\mathbf{x}|\omega_i\right)$ is the class-conditional probability density function for $\mathbf{x}$, or the likelihood of having the pattern $\mathbf{x}$ given an object that belongs to class ω_i. Having the pattern $\mathbf{x}$, which is an observed value for $\mathbf{X}$, and knowing the prior probability for all the states of nature, $P\left(\omega_j\right)$, and the class-conditional densities for $\mathbf{x}$, the probability of the state of nature being ω_j can be calculated, using *Bayes' formula*:

$$P\left(\omega_j|\mathbf{x}\right) = \frac{p\left(\mathbf{x}|\omega_j\right)P\left(\omega_j\right)}{p\left(\mathbf{x}\right)} \tag{1.1}$$

where

$$p\left(\mathbf{x}\right) = \sum_{j=1}^{c} p\left(\mathbf{x}|\omega_j\right)P\left(\omega_j\right) \tag{1.2}$$

being c the number of different possible states of nature. $P\left(\omega_j|\mathbf{x}\right)$ is called the posterior probability.

Notice that it is the product of the likelihood and the prior probability that is the most important determining the posterior probability, the *evidence factor*, $p\left(\mathbf{x}\right)$, can be seen just as a scale factor.

Moreover, Naive Bayes classification assumes that the variables in $\mathbf{X}$ are independent. By making that assumption, the class-conditional probability can be simplified and written in the following form:

$$p\left(\mathbf{x}|\omega_i\right) = \prod_{i=1}^{m} p\left(x_i|\omega_i\right) \tag{1.3}$$

Definition 1.2.2. (Loss function) Let $\{\alpha_1, \alpha_2, \ldots, \alpha_C\}$ be the finite set of possible actions, where taking the action α_i means that $\hat{z} = i$, being $\hat{z}$ the label estimated by the classifier for a given pattern. The *loss function*, $\lambda\left(\alpha_i|\omega_j\right)$, is a function that states how costly each decision is. It describes the loss incurred by taking the action α_i, i.e., deciding that the class for pattern $\mathbf{x}$ is ω_i when the state of nature is ω_j.

Definition 1.2.3. (Conditional risk) Using the previous definition, the *conditional risk* can be defined. Given a particular pattern $\mathbf{x}$ and assuming that the decision of the classifier is $\hat{z} = i$, that is, the action taken is α_i, then the conditional risk incurred will be

$$R\left(\alpha_i|\mathbf{x}\right) = \sum_{j=1}^{c} \lambda\left(\alpha_i|\omega_j\right)P\left(\omega_j|\mathbf{x}\right) \tag{1.4}$$

The aim will be to find a decision rule that minimizes the overall risk.

Definition 1.2.4. (Decision rule) A *decision rule* is a function α from the set of all possible patterns to the set of all possible actions $\{\alpha_1, \alpha_2, \ldots, \alpha_C\}$. $\alpha(\mathbf{x})$ takes one of the possible values from $\{\alpha_1, \alpha_2, \ldots, \alpha_C\}$, in other words, it is the function that states which action to take for every possible pattern.

Definition 1.2.5. (Overall risk) Let α be a decision rule, since $R(\alpha_i|\mathbf{x})$ is the conditional risk associated with α_i, the *overall risk* will be given by

$$R = \int R(\alpha(\mathbf{x})|\mathbf{x})\, p(\mathbf{x})\, d\mathbf{x} \tag{1.5}$$

Then, $\alpha(\mathbf{x})$ is chosen so that $R(\alpha_i(\mathbf{x}))$ is as small as possible for every $\mathbf{x}$.

This leads to the statement of the *Bayes decision rule*: To minimize the overall risk compute the conditional risk $R(\alpha_i|\mathbf{x})$ for $i = 1, \ldots, a$ and select the action α_i for which it is minimum. The resulting overall risk is called Bayes risk, denoted R^*, and it is the best performance that can be achieved. [1]

1.2.1 Two-category Classification

Let us consider the case in which there are only two categories. Then, action α_1 corresponds to deciding that the true state of nature is ω_1, and action α_2 corresponds to deciding that it is ω_2. If $\lambda_{ij} = \lambda(\alpha_i|\omega_j)$ then the conditional risk can be written as

$$\begin{aligned} R(\alpha_1|\mathbf{x}) &= \lambda_{11}P(\omega_1|\mathbf{x}) + \lambda_{12}P(\omega_2|\mathbf{x}) \text{ and} \\ R(\alpha_2|\mathbf{x}) &= \lambda_{21}P(\omega_1|\mathbf{x}) + \lambda_{22}P(\omega_2|\mathbf{x}) \end{aligned} \tag{1.6}$$

As the decision rule would be to decide ω_1 if $R(\alpha_1|\mathbf{x}) < R(\alpha_2|\mathbf{x})$, in terms of probabilities it can be seen as deciding ω_1 if

$$(\lambda_{21} - \lambda_{11})\, P(\omega_1|\mathbf{x}) > (\lambda_{12} - \lambda_{22})\, P(\omega_2|\mathbf{x}) \tag{1.7}$$

Using Bayes' formula to replace posterior probabilities by prior probabilities and assuming that $\lambda_{21} > \lambda_{11}$ (as the loss of being correct will be smaller than the loss of failing), the next equivalent rule is obtained: decide ω_1 if

$$\frac{p(\mathbf{x}|\omega_1)}{p(\mathbf{x}|\omega_2)} > \frac{(\lambda_{12} - \lambda_{22})\, P(\omega_2)}{(\lambda_{21} - \lambda_{11})\, P(\omega_1)} \tag{1.8}$$

The left term at (1.8) is called *likelihood ratio.* Thus, the Bayes decision rule can be seen as deciding for ω_1 if the likelihood ratio exceeds from a value that does not depend on $\mathbf{x}$.

1.2.2 Minimum-Error-Rate Classification

Each state of nature is usually associated with a different one of the c classes. If α_i is the action taken and ω_j the true state of nature, then the decision is correct if $i = j$ and it is an error if $i \neq j$. It is natural to seek a decision rule that minimizes the probability of error, i.e., the error rate.

Definition 1.2.6. (Symmetrical loss function) The *symmetrical* or *zero-one loss function* is a loss function defined in the following way:

$$\lambda\left(\alpha_i|\omega_j\right) = \begin{cases} 0 & i = j \\ 1 & i \neq j \end{cases} \tag{1.9}$$

In this way, the symmetric loss function assigns no loss to a correct decision and a unit loss to any error.

When the symmetrical loss function is used, the conditional risk can be written as follows:

$$R\left(\alpha_i|\mathbf{x}\right) = \sum_{j=1}^{c} \lambda\left(\alpha_i|\omega_j\right) P\left(\omega_j|\mathbf{x}\right) = \sum_{j\neq i} P\left(\omega_j|\mathbf{x}\right) = 1 - P\left(\omega_i|\mathbf{x}\right) \tag{1.10}$$

Then, to minimizing the conditional risk is equivalent to maximizing the posterior probability. This means that the decision rule will be to decide ω_i if $P\left(\omega_i|\mathbf{x}\right) > P\left(\omega_j|\mathbf{x}\right)$ for all $j \neq i$.

Remark 1.2.1. Notice that the symmetrical loss function gives the same importance to all the missclassifications. However, it can happen that one mistake is worse that other, for example, imagine the case in which blood tests are carried out to diagnose an infection. Classifying a healthy person as ill and prescribing she antibiotics she does not need could be bad, but having an ill person classified as healthy and sent home can lead into serious worsening and even death.

Thus, it is important to know in each case what is the cost of the different mistakes, and to choose an appropriate loss function that represents it.

1.2.3 Classifiers, Discrimination Functions and Decision Surfaces

Another way to define a classifier is through discriminant functions.

Definition 1.2.7. (Discriminant functions) A function g_i from a set of functions $\{g_j\}_{j=1,\dots,C}$ is called a *discriminant function* if given a pattern $\mathbf{x}$ the decision taken is

$$\mathbf{x} \text{ belongs to } \omega_i \Leftrightarrow g_i(\mathbf{x}) > g_j(\mathbf{x}) \forall j \neq i$$

[4]

Thus, the classifier computes the C discriminant functions and selects the category corresponding to the greatest result.

In general, discriminant functions for Bayesian classifiers are defined as $g_i(\mathbf{x}) = -R(\alpha_i|\mathbf{x})$, so that the maximum discriminant function corresponds to the minimum conditional risk. In this way, the way of choosing the class for a pattern is the same as the one stated in the Bayes decision rule. However, for the minimum-error-rate case discriminant functions can simply be taken as $g_i(\mathbf{x}) = P(\omega_i|\mathbf{x})$.

Notice that the choice of the discriminant functions is not unique; adding or multiplying with a constant (that must be positive in case of multiplication) or even applying a monotonically increasing function to every g_i, does not change the classification.

Through those discriminant functions, the feature space is divided into *c decision regions*, $\mathcal{R}_1, \ldots, \mathcal{R}_c$. If $g_i(\mathbf{x}) > g_j(\mathbf{x})$ for all $j \neq i$, then $\mathbf{x}$ is in $\mathcal{R}_i$ and the decision rule assigns $\mathbf{x}$ to ω_i. The regions are separated by decision boundaries, which are surfaces in the feature space where the values of the largest discriminant functions are the same. It is necessary to correctly adjust the decision boundary, if it is too simple it may not distinguish properly among two categories. But if the decision boundary is to complex the classifier may work perfectly in the training set but make more mistakes with new elements, because it provides a non-general classification, too adjusted to the training set.

1.2.4 Bayes Decision Theory - Discrete Features

When the feature vector $\mathbf{x}$ only takes discrete values, the probability density function $p(\mathbf{x}|\omega_j)$ becomes singular and then integrals of the form

$$\int p(\mathbf{x}|\omega_j)\, d\mathbf{x} \tag{1.11}$$

must be replaced by

$$p(\mathbf{x}|\omega_j) = \sum_{\mathbf{x}} P(\mathbf{x}|\omega_j) \tag{1.12}$$

Then Bayes' formula is expressed using probabilities instead of probability densities:

$$P(\omega_j|\mathbf{x}) = \frac{P(\mathbf{x}|\omega_j)\, P(\omega_j)}{P(\mathbf{x})} \tag{1.13}$$

where

$$P(\mathbf{x}) = \sum_{j=1}^{c} P(\mathbf{x}|\omega_j)\, P(\omega_j) \tag{1.14}$$

The definition of conditional risk $R(\alpha|\mathbf{x})$ is unchanged, as well as the fundamental Bayes decision rule. Also the discriminant functions remain the same, except for the replacement of densities $p(\cdot)$ by probabilities $P(\cdot)$.

Chapter 2

Dimensionality reduction

This chapter presents the issue of dimensionality reduction, explaining its benefits and the different techniques used for the reduction. Some of the most common methods are also explained.

2.1 Importance of dimensionality reduction

The complexity of a classifier depends strongly on the number of inputs that it receives. If the input is a feature vector with many dimensions, the classifier will be more complex than if the feature vector has lower dimensionality. This is why dimensionality reduction is a very important and interesting part of the process of pattern recognition.

By reducing dimensionality not only complexity of the classifier is reduced, other important advantages also derive from having less dimensions in the feature space:

- Sometimes, if an input is not necessary at all for the classification, it can be completely removed and, thus, in the future that feature does not need to be measured. That is, the cost of extracting some features can be saved.
- With simpler classifiers an improved performance is achieved, as simpler models are more stable and have less variance. That means that a simpler model will be less depending on factors as noise or the occurrence of outliers.
- In some cases, the dimensions can be reduced enough to make it possible to plot the data and analyze it visually. The advantage of the visual analysis of the data is that it helps to discover underlying structures.

In conclusion, reducing the number of variables can lead to an improved classifier performance and a grater understanding of the data. Thus, different ways of achieving dimensionality reduction have been studied. Given a set of variables, dimensionality reduction methods can be separated in two broad categories: feature selection and feature transformation.

On the one hand, *feature selection* consists on identifying the variables that do not contribute to the classification task and discarding them. Out of the total m dimensions only the k that carry most information will be chosen, where k is a number to be determined.

On the other hand, *feature transformation* is the process of finding a transformation from the original m variables into a new set of k variables, being k less than or equal to m.

In fact, also feature selection methods can be seen as part of feature transformation. While doing the transformations a set of weights is applied to the original variables to obtain the new ones, feature selection is the result of using binary weights for that transformation, instead of continuous [5, Chapter 6] [3, Chapter 10]. However, in this work those two categories will be studied separately.

2.2 Feature selection

Feature selection consists on choosing a feature subset in the total feature space, out of the possible m feature dimensions only k will be chosen. This k features need to be the best possible feature subset, that is, it must be the subset that most contributes to accuracy with the least number of dimensions. The rest of the features will be discarded and will not be used for classification.

2.2.1 Relevance of a feature

It is common that a large number of features are not informative, because they are irrelevant or redundant. Given a feature set $\mathcal{X}$ and a feature X in $\mathcal{X}$, S will denote the set of features excluding X. Then, the relevance of the feature X can be one of the following four:

Definition 2.2.1. (Strong relevance)
The feature X is *strongly relevant* if and only if $P\left(\hat{z}|X,S\right) \neq P\left(\hat{z}|S\right)$. That is, the distribution of the class predictor depends on the feature X.

Definition 2.2.2. (Weak relevance)
The feature X is *weakly relevant* if and only if $P\left(\hat{z}|X,S\right) = P\left(\hat{z}|S\right)$ and $\exists S' \subset S$ such that $P\left(\hat{z}|X,S'\right) \neq P\left(\hat{z}|S'\right)$. This means that removing X

from the total set does not affect the class prediction but there exists a subset of features for which it does.

Definition 2.2.3. (Irrelevance)
The feature X is *irrelevant* if and only if $\forall S' \subseteq S, P(\hat{z}|X, S') = P(\hat{z}|S')$. That is, X is not necessary for class prediction.

Definition 2.2.4. (Redundancy)
To define redundancy Markov Blanket needs to be defined first.

> *Markov blanket:* Let $\mathbf{M}_X$ be a subset of the features $\mathcal{X}$ that does not contain X. $\mathbf{M}_X$ is a Markov blanket of X if X is conditionally independent of $\mathcal{X}-\mathbf{M}_X-\{X\}$ given $\mathbf{M}_X$. That is, if $P(X, \mathcal{X} - \mathbf{M}_X - \{X\}|\mathbf{M}_X) = P(X|\mathbf{M}_X)\,P(\mathcal{X} - \mathbf{M}_X - \{X\}|\mathbf{M}_X)$. This means that all the information that X provides about other features is contained in $\mathbf{M}_X$.

Given $G \subset \mathcal{X}$ a set of features, a feature in G is redundant if and only if it is weakly relevant and has a Markov blanket in G.

The goal of feature selection is to choose an optimal feature set, and this will be a feature set that contains only strongly relevant and some weakly relevant features.

2.2.2 Feature selection approach

With 2^m possible subsets of variables, being m the total number of features, is it impossible (or at least not efficient) to test all of them. Thus, different techniques exist for feature selection.

A subset selection method is composed of two steps: *subset generation*, that is, choosing a feature subset, and *evaluation*, measuring how "good" the selected subset is. The process can be recursive: a subset is chosen, then it is evaluated and the information obtained is used again to choose another subset. Image 2.1 is an scheme of the complete process of feature selection.

Thus, each possible feature selection method is a combination of one of the possible evaluation measures and one of the subset search algorithms. The methods are usually classified in one of the following three categories: filter methods, wrapper methods and embedded methods.

Filter methods

This kind of methods use statistical properties of the variables to filter the ones that provide less information. Feature selection is done before classifi-

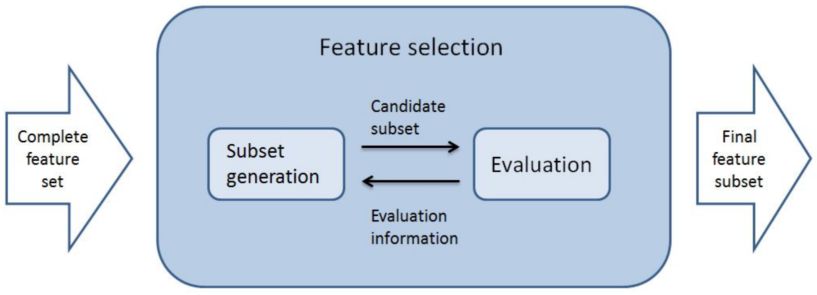

Figure 2.1: Scheme of the process of Feature Selection showing the interaction between subset generation and subset evaluation. Image based on a scheme found in [2]

cation, and therefore is independent from the classifier used.

One example of a filter method is the *Variance Threshold* method, that consist on removing the features with a variance lower than a fixed threshold. In the last years also Markov blankets have been used as a filter method, by searching for the Markov blanket of the class variable. This would be a minimal set of variables such that all other variables are conditionally independent of the class variable given the variables in the Markov blanket [3].

Wrapper methods

These ones are classifier dependent since subsets are evaluated within the classification process, being the performance of the classifier the measure used to evaluate the subset. They usually have better performance than filter methods but are more computationally demanding.

Embedded methods

These are also classifier dependent, they can be seen as the search in the combined space of feature subsets and classifier models. The decision tree classifier or the Weighted Naive Bayes classifier are examples of embedded methods.

2.2.3 Evaluation measures

In order to choose a good feature set, it is necessary to somehow measure the ability of the candidate set to contribute to the separation of the classes. Features in the candidate are evaluated either individually or in the context of the other features. There are mainly two types of measures, the ones that rely on the properties of the data and the ones that use a classification rule as part of the evaluation.

The first ones are used in filter methods, as they only focus on statistical properties of the variables. The ones that use a classification rule are used in wrapper and embedded classification methods. In this second approach a classifier is designed using measurements in the candidate subset. Then, the performance of the classifier is assessed using some of the metrics described in Section 1.1.3 and the obtained value is used as subset evaluation measure. However, in this case the performance of the classifier is not evaluated using the test set, but a set called *validation set* which is part of the training set. Otherwise, the features chosen would be the the ones that fit specially well the examples on the test set, and the resulting classifier would be very good for that specific set but not able to generalize. Thus, the chosen feature set will be the one for which the classifier performs well on a separate validation set.

Among the measures that relay on the properties of the data, four different categories can be differentiated: feature ranking measures, interclass distance measures, probabilistic distance measures and probabilistic dependence measures. All of those four families of evaluation criteria are independent of the classifier used and therefore they are usually cheaper to implement than the ones that are classifier dependent. However, the estimation is also poorer than the one involving classifiers.

Feature ranking

This method is used to rank features individually, it is a simple method that allows removing irrelevant or redundant features. They are easy to calculate, and the simplest ones are the ones based in correlation.

The *Pearson correlation* coefficient is used to measure the linear correlation among two variables. Given the variables X and Y with measurements $\{x_i\}$ and $\{y_i\}$ and means $\bar{x}$ and $\bar{y}$ the Pearson correlation coefficient is given by

$$\rho(X,Y) = \frac{\sum_{i\in I}(x_i - \bar{x})(y_i - \bar{y})}{\left[\sum_{i\in I}(x_i - \bar{x})^2 \sum_{i\in I}(y_i - \bar{y})^2\right]} \tag{2.1}$$

When $\rho(X,Y) = \pm 1$ then X and Y are completely correlated, so one of them is redundant and can be eliminated.

However, when relations among features are nonlinear, a nonlinear correlation measure is needed. This is the case of the *mutual information*. Given a discrete variable X its entropy is defined as

$$H(X) = -\sum_x p(x) log_2(p(x)) \tag{2.2}$$

and the entropy of X after observing Y is defined as

$$H(X|Y) = -\sum_y p(y) \sum_x p(x|y) log_2(p(x|y)) \tag{2.3}$$

Then the mutual information is the additional information about X provided by Y and it represents the decrease in the entropy of X that occurs when Y is observed

$$MI(X|Y) = H(X) - H(X|Y) \tag{2.4}$$

A normalized value of the mutual information can also be given. It is called *symmetrical uncertainty* and is given by

$$SU(X,Y) = 2\left(\frac{MI(X|Y)}{H(X) + H(Y)}\right) \tag{2.5}$$

This measure gives a value between zero and one, with zero meaning that the features are independent and one that they are completely correlated.

Mutual information and symmetrical uncertainty allow nonlinear dependence among variables to be detected, and work with discrete features. However, their disadvantages are that continuous features must be discretized and that probability density functions must be estimated.

All those measures can be used to rank features individually according to their relevance. However, for the features that become relevant only in the context of others, a ranking criteria that takes context into account is needed. So, there exist measuring methods that take context into consideration, for example, the family of algorithms called *Relief* [3].

Interclass distance

Interclass distance measures the distance between classes based on the distances between members of each class.

Given a measure of distance, $d(\mathbf{x}, \mathbf{y})$, between two patterns of different classes ($\mathbf{x} \in \omega_1$ and $\mathbf{y} \in \omega_2$), a measure of the separation between the classes ω_1 and ω_2 is defined as

$$J_{as}(\omega_1, \omega_2) = \frac{1}{n_1 n_2} \sum_{i=1}^{n_1} \sum_{j=1}^{n_2} d(\mathbf{x}_i, \mathbf{y}_j) \tag{2.6}$$

Dissimilarity measure	Mathematical form
Divergence	$J_D(\omega_1, \omega_2) = \int [p(\mathbf{x}\|\omega_1) - p(\mathbf{x}\|\omega_2)] \log\left(\frac{p(\mathbf{x}\|\omega_1)}{p(\mathbf{x}\|\omega_2)}\right) d\mathbf{x}$
Chernoff	$J_c(\omega_1, \omega_2) = -\log \int p^s(\mathbf{x}\|\omega_1) p^{1-s}(\mathbf{x}\|\omega_2) d\mathbf{x}$
Bhattacharyya	$J_B(\omega_1, \omega_2) = -\log \int (p(\mathbf{x}\|\omega_1) p(\mathbf{x}\|\omega_2))^{\frac{1}{2}} d\mathbf{x}$
Patrick-Fischer	$J_P(\omega_1, \omega_2) = \left(\int [p(\mathbf{x}\|\omega_1)p(\omega_1) - p(\mathbf{x}\|\omega_2)p(\omega_2)]^2\right)^{\frac{1}{2}}$

Table 2.1: Probabilistic distance measures

being n_1 and n_2 the number of patterns belonging to class ω_1 and ω_2, respectively. For more than two classes the average distance between classes is defined as

$$J = \frac{1}{2} \sum_{i=1}^{C} p(\omega_i) \sum_{j=1}^{C} p(\omega_j) J_{as}(\omega_i, \omega_j) \tag{2.7}$$

where $p(\omega_i)$ is the prior probability of class ω_i (estimated as $p_i = n_i/n$). Notice that this is simply the total variance, which is not the best criterion for feature selection. Other, more efficient ways to define J can be found in [3].

There exist may different ways to define the distance between two patterns, $d(\mathbf{x}, \mathbf{y})$, that depend on the type of variable. For example, for binary variables Russel and Rao[1] distance can be defined and for numeric variables Euclidean distance[2] or Chebyshev distance[3] could be used.

Probabilistic distance

Probabilistic distance measures are used in the case of having only two categories, as they measure the distance between two distributions, $p(\mathbf{x}|\omega_1)$ and $p(\mathbf{x}|\omega_2)$. Some of the most used measures are given in table 2.1. Those measures will reach a maximum when the classes are disjoint. Their disadvantage is the need to estimate the probability density function and its integral. However, it is common that under certain assumptions the expressions can be evaluated analytically.

Probabilistic dependence measures

Probabilistic dependence measures are equivalent to probabilistic distance measured but in this case for multiclass problems. They are used to cal-

[1] d_{rr} = number of occurrences of $x_i = 1$ and $y_i = 1$ / total number of occurrences

[2] $d_e = \left[\sum_{i=1}^{p} (x_i - y_i)^2\right]^{\frac{1}{2}}$

[3] $d_{ch} = \max_i |x_i - y_i|$

Dissimilarity measure	Mathematical form
Divergence	$J_D = \sum_{i=1}^{C} p(\omega_i) \int [p(\mathbf{x}\vert\omega_i) - p(\mathbf{x})] \log \left(\frac{p(\mathbf{x}\vert\omega_i)}{p(\mathbf{x})}\right) d\mathbf{x}$
Chernoff	$J_c = \sum_{i=1}^{C} p(\omega_i) \left(- \log \int p^s(\mathbf{x}\vert\omega_i) p^{1-s}(\mathbf{x}) d\mathbf{x}\right)$
Bhattacharyya	$J_B = \sum_{i=1}^{C} p(\omega_i) \left(- \log \int (p(\mathbf{x}\vert\omega_i) p(\mathbf{x}))^{\frac{1}{2}} d\mathbf{x}\right)$
Patrick-Fischer	$J_P = \sum_{i=1}^{C} p(\omega_i) \left(\int [p(\mathbf{x}\vert\omega_i) - p(\mathbf{x})]^2\right)^{\frac{1}{2}}$

Table 2.2: Probabilistic dependence measures

culate the distance between the class-conditional density, $p(\mathbf{x}|\omega_i)$, and the probability density function, $p(\mathbf{x})$. If those two are identical, there is no class information gain by observing $\mathbf{x}$. And if they are very different, then the dependence of $\mathbf{x}$ on the class ω_i is high. Table 2.2 gives the probabilistic dependence measures corresponding to table 2.1. In practice, probabilistic dependence measures are difficult to apply, because the expressions given in table 2.2 cannot be evaluated analytically.

2.2.4 Search algorithms for feature subset selection

There exist three categories of search algorithms: complete, sequential and random search.

Complete search

Complete search methods are the ones that guarantee to find the optimal subset. It is a complete search method, for instance, the *exhaustive search*, that is, evaluating all the possible feature subsets and selecting the best one. However, this method is extremely costly, and so other ones that do not need the evaluation of all the possible subsets are sought.

One complete search method that does not require the evaluation of all the possible subsets is, for example, *Branch and bound* method. Branch and bound consists in the creation of a tree in which the nodes represent all the possible feature subsets. In each level the nodes contain a set with features that where on the previous node with one of them removed. The method assumes the monotonicity property, that is, if A and B are two feature subsets such that $A \subset B$, and the function J represents the chosen evaluation measure, for Branch and bound method to work it is needed that $J(A) < J(B)$ (where a higher value for J means a better performance).

The algorithm starts at the least dense parts of the tree, evaluating a feature subset there. Then it backtracks and goes down other branches until it finds a subset worse that the best subset found yet. In that case it will again backtrack and go down another branch. Thus, although not all subsets will be evaluated, the optimal one will be found.

Sequential search

In sequential search features are added or removed sequentially. This methods do not always find the optimal subset, they may get stuck in local optimal. However, they are simple to implement and produce fast results.

Let S be a feature subset of all the possible features $S = \{X_i\}_{i=1}^{d}$ where $d \leq m$, being m the number of available features to chose from. And let the function J represent the chosen evaluation measure for the feature subset. $J(S)$ is the value of J on the validation sample when only inputs in S are used (J is checked on a validation set different from the training set to test the generalization) [5]. Then the following methods describe different types of sequential search algorithms.

Best individual d

This is the simplest way of choosing d features. Each of the possible m features is assigned a score through the evaluation measure J and the features are ordered so that:

$$J(X_1) \geq J(X_2) \geq \cdots \geq J(X_m)$$

Then, the first d features are selected as the feature subset, $S = \{X_i\}_{i=1}^{d}$. Notice that this method needs to know in advance how many features are going to be selected, d. It can produce good subsets but it is not the most efficient method, specially when features are highly correlated [3].

There exists also a variation of this method in which, instead of choosing a number d of features to keep, a percentage k is chosen. Thus, the number d of features to maintain will be k% of the total number of features.

Sequential forward selection

This method starts with a subset with 0 features, $S = \emptyset$. Then, features will be added one by one, choosing at each time the one that improves the value $J(S)$. For example, assuming that the measure J is the error rate E the chosen feature will be the one that decreases the error most. This will go on until any further addition does not decrease the error (or decreases it only slightly).

That is, for all possible X_i, calculate $E(S \cup X_i)$ and choose the input X_j that causes least error $j = \arg\min_i E(S \cup X_i)$. If $E(S \cup X_j) < E(S)$ then add X_j to S. If adding does not decrease E (or decreases it very little) the search stops.

This method can be costly, as the system has to be trained and tested $m + (m-1) + (m-2) + ... + (m-k)$ times, $O(m^2)$. Moreover, it performs a local search, it does not guarantee finding the optimal subset. Another disadvantage is that it adds features one by one so, if X_i and X_j are good together but not by themselves, it may not be able to detect it.

It is also possible to add more that one feature at a time, with the expense of some more computation. This is a variation of the method called Generalized sequential forward selection.

Remark 2.2.1. To explain sequential forward selection the evaluation measure J used has been E, the error rate. In this case is desirable to have a small error rate. If other measure was used such that higher value represents better performance (such as Accuracy or F1 value) then the chosen feature would be the one that most increases the value of J, and the search would go on until J stopped increasing.

While explaining the Sequential backward selection error rate will be used again as an example of evaluation measure. However, it is important to keep in mind that with another kind of evaluation measure, "increase" and "grater" may need to be substituted with "decrease" and "smaller", or vice versa.

Sequential backward selection

In this case, the initial feature subset S contains all the features. Then, they are removed one by one, each time choosing to remove the one that makes the error decrease the most. It may also be allowed to remove a feature if that action increases the error only a slightly, because, even if the error is increased, removing one more feature will decrease complexity.

That is, for all possible X_i, calculate $E(S - X_i)$ and choose the input X_j that causes least error $j = \arg\min_i E(S - X_i)$. If $E(S - X_j) < E(S)$ then remove X_j from S. If removing X_j increases the error considerably, the search stops.

Complexity of sequential backward selection is the same as in forward, except that training the system with more features is more costly. For this reason, if many features are expected to be useless, it is better to use sequential forward selection.

Also in this case, more than one feature can be removed at each time, in what is known as *generalized sequential backward selection.*

Bidirectional search

There is a way of using both sequential forward selection and sequential backward selection and it is called bidirectional search.

In this method sequential forward and backward selection are performed in parallel, each one starting, correspondingly, from $S = \emptyset$ and S the full feature set. Features already selected by sequential forward selection are not removed by sequential backward selection, and features already removed by sequential backward selection are not selected by sequential forward selection. In this way, it is guaranteed that both methods converge to the same solution.

Plus-l minus-r selection

This is another type of procedure in which both sequential forward selection and sequential backward selection are used.

In this case, l features are added by sequential forward selection and then r removed using sequential backward selection. It is similar to bidirectional search but in this case feature adding and removing are not done in parallel, one goes after the other.

If $l > r$ it starts adding l features with sequential forward selection and then removes r with sequential backward selection. Conversely, if $r > l$ it will start by removing features.

When the number of features added and removed, l and r change in each iteration of the search procedure, this is called a *floating search.*

Randomized search

When randomized search is carried out, feature subsets are randomly generated or randomness is included in the previous methods. This is very useful when the space of possible feature subsets is large. It also helps to avoid getting trapped in local optimal subsets, and the benefits of obtaining a good solution are more than the computational costs of including randomness.

2.3 Feature transformation

Feature transformation consists on the transformation of the original data into a lower-dimensional data sets. Unlike in feature selection methods, feature transformation uses all of the original variables and transforms them into a smaller set of underlying variables. The transformations studied will be data adaptive, that is, depending on the data, and both supervised or unsupervised.

2.3.1 Principal component analysis (PCA)

PCA is the most used feature extraction method. Geometrically, principal components analysis can be seen as a rotation of the axes into a new set of orthogonal axes that are ordered by the amount of variation of the original data in each of the directions of the new axes. Then, the first few new directions or *components* will contain almost all the variation of the original data. Thus, the fist components will be kept while the other ones are discarded, obtaining in this way an smaller set of variables that, even if they may not have an interpretation, describe most of the variation of the original data.

In that sense, PCA can also be seen as a projection method, where the inputs in the original d dimensional space are projected into a new k ($k < d$) dimensional space with the minimum loss of information. It is an unsupervised method, that does not use the output information, so the criterion to be maximized is the variance of the data in the new direction. If $\mathbf{x}$ is the original data, the aim will be to project it in the direction ω such that the projection of $\mathbf{x}$ in the direction of ω, $\mathbf{z} = \omega^T\mathbf{x}$, is the most spread out, has the most variance.

The vector ω_1 that maximizes the variance of $\mathbf{z}_1 = \omega_1^T\mathbf{x}$ will be called *principal component*. In order to calculate it, $\|\omega_1\| = 1$ will be required, so that an unique solution is achieved and the direction becomes the important factor in the choosing of the principal component. Then, as the variance of $\mathbf{z}_1$ is,

$$\begin{aligned}
\mathrm{Var}(\mathbf{z}_1) &= \mathrm{E}[\mathbf{z}_1^2] - \mathrm{E}[\mathbf{z}_1]^2 && (2.8)\\
&= \mathrm{E}[\omega_1^T\mathbf{x}\mathbf{x}^T\omega_1] - \mathrm{E}[\omega_1^T\mathbf{x}]\mathrm{E}[\mathbf{x}^T\omega_1] && (2.9)\\
&= \omega_1^T\mathrm{E}[\mathbf{x}\mathbf{x}^T]\omega_1 - \omega_1^T\mathrm{E}[\mathbf{x}]\mathrm{E}[\mathbf{x}^T]\omega_1 && (2.10)\\
&= \omega_1^T(\mathrm{E}[\mathbf{x}\mathbf{x}^T] - \mathrm{E}[\mathbf{x}]\mathrm{E}[\mathbf{x}^T])\omega_1 && (2.11)\\
&= \omega_1^T\boldsymbol{\Sigma}\omega_1 && (2.12)
\end{aligned}$$

where $\boldsymbol{\Sigma}$ is the covariance matrix of $\mathbf{x}$ and E[.] denotes expectation.

A ω_1 such that $\mathrm{Var}(\mathbf{x}_1)$ is maximized while $\omega_1^T\omega_1 = 1$ is wanted. This problem can be written as a Lagrange problem in the following way:

$$\max_{\omega_1} \omega_1^T \mathbf{\Sigma} \omega_1 - \alpha \left(\omega_1^T \omega_1 - 1 \right) \tag{2.13}$$

And taking the derivative with respect to ω_1 and setting it equal to 0:

$$2\mathbf{\Sigma}\omega_1 - 2\alpha\omega_1 = 0 \Leftrightarrow \mathbf{\Sigma}\omega_1 = \alpha\omega_1 \tag{2.14}$$

Thus, equation 2.14 is hold if ω_1 is an eigenvector of $\mathbf{\Sigma}$ and α is the eigenvalue that corresponds to ω_1. On the other hand, from equation 2.14 also follows that:

$$\mathbf{\Sigma}\omega_1 = \alpha\omega_1 \Rightarrow \omega_1^T\mathbf{\Sigma}\omega_1 = \alpha\omega_1^T\omega_1 \tag{2.15}$$

Taking into account that $\omega_1^T\omega_1 = 1$ and equality 2.12, it follows that

$$\mathrm{Var}(\mathbf{z}_1) = \alpha \tag{2.16}$$

where α is an eigenvalue of $\mathbf{\Sigma}$. Thus, for the variance to be maximum, α is chosen as the largest eigenvalue, which implies that ω_1 is the eigenvector corresponding to the largest eigenvalue [5] [3].

Similarly, the second principal component turns out to be the eigenvector with the second largest eigenvalue. In this way, the directions that maximize the variance are given by the eigenvectors of the sample covariance matrix for an observed $\mathbf{x}$, ordered by their corresponding eigenvalues in a decreasing order.

Once the eigenvalues are ordered, the first k have to be selected an the other ones discarded. But k is a number chosen by the user, and it is very much problem specific, making it difficult to select an appropriate value for it. One way of choosing k is choosing the *proportion of variance* that needs to be explained after discarding the rest of eigenvalues.

Definition 2.3.1. (Proportion of variance) When the eigenvectors λ_i of the sample covariance matrix for an observed $\mathbf{x}$ are sorted in descending order, the *proportion of variance* explained by the k principal components is

$$\frac{\lambda_1 + \lambda_2 + \cdots + \lambda_k}{\lambda_1 + \lambda_2 + \cdots + \lambda_k + \cdots + \lambda_d} \tag{2.17}$$

In this way, the proportion of variance to remain explained can be chosen, for example, a 90% of the variance. Then, the k eigenvalues that explain the 90% of the variance will be kept. [5] However, the user still needs to chose the proportion of variance with this method. Another approach is to analyze the *Scree graph*, which is the plot of the ordered eigenvalues as a function of the eigenvalue number. This graph usually falls sharply before levelling

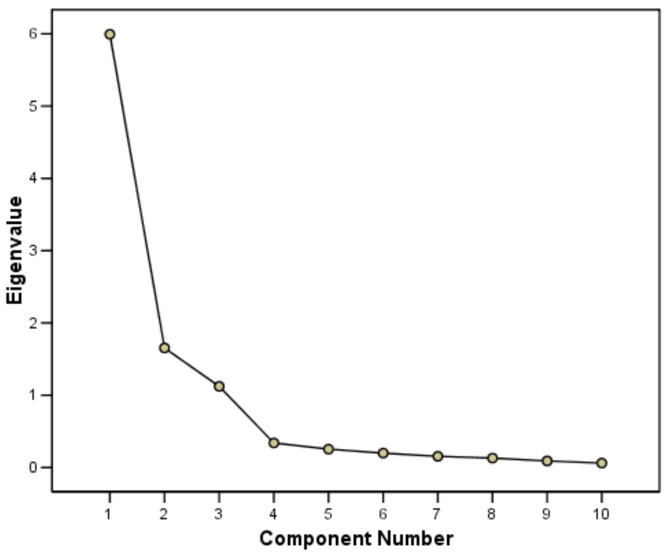

Figure 2.2: Example of an scree graph with an elbow at the fourth component. Source: http://www.ibm.com/support/knowledgecenter/SSLVMB_ 21.0.0/com.ibm.spss.statistics.cs/fac_ cars_ scree_ 01.htm

off at small values (see figure 2.2), creating an "elbow" that is used as the cutting point to choose the eingenvectors to keep [3]. Still, this method is not always reliable either, as the scree graph is not always of that form.

Remark 2.3.1. Before performing PCA transformation, it is recommended to preprocess the data do that it has 0 mean and variance 1, otherwise the units in which the different features are measured could alter directions of the principal components [3]. PCA is also sensitive to outliers in the observed data.

Remark 2.3.2. PCA is an unsupervised feature transformation method, it does not use class information. However, there exist different variants of this method, called Karhunn-Loève expansions, that allow using class information (see [3]).

2.3.2 Linear Discriminant Analysis

Linear Discriminant Analysis (LDA) is a supervised method for dimensionality reduction. LDA consists on finding new directions such that, when the data is projected in that directions, elements of the same class are next to each other but far from elements of other classes.

Definition 2.3.2. (Scatter matrices) Given a set of patterns and label vectors $\mathcal{X} = \{\mathbf{x}^t, \mathbf{r}^t\}$ such that $r_i^t = 1$ if $\mathbf{x}^t \in C_i$ and $r_i^t = 0$ otherwise, the scatter matrices are defined in the following way:

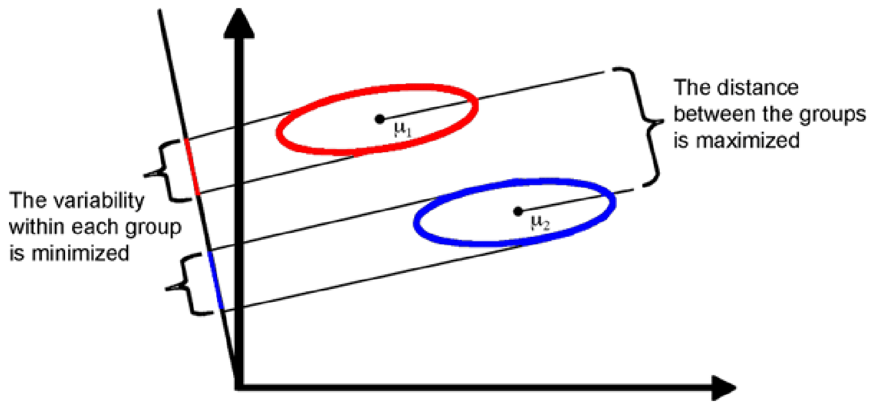
The distance between the groups is maximized
The variability within each group is minimized
μ_1
μ_2

Figure 2.3: Example of LDA projection for a two-dimensional two-class problem. Source: http://compbio.pbworks.com/w/page/16252905/Microarray%20Dimension%20Reduction

- *Within-class scatter matrix:* $\mathbf{S}_i = \sum_t r_i^t(\mathbf{x}^t - \mathbf{m}_i)(\mathbf{x}^t - \mathbf{m}_i)^T$
- *Total within-class scatter matrix:* $\mathbf{S}_W = \sum_{i=1}^k \mathbf{S}_i$
- *Between-class scatter matrix:* $\mathbf{S}_B = \sum_{i=1}^k N_i(\mathbf{m}_i - \mathbf{m})(\mathbf{m}_i - \mathbf{m})^T$

  where:
  - $N_i = \sum_t r_i^t$
  - $\mathbf{m}_i :=$ mean of sample from class C_i
  - $\mathbf{m} = \frac{1}{K}\sum_{i=1}^K \mathbf{m}_i$

If the original data has m dimensions and the projection is going to be done into new k dimensions, the aim of LDA is to find $\mathbf{W}$, a $m \times k$ matrix such that the between-class scatter after projection, $\mathbf{W}^T\mathbf{S}_B\mathbf{W}$, is large and the within-class scatter after projection, $\mathbf{W}^T\mathbf{S}_W\mathbf{W}$, is small. Thus, $\mathbf{W}$ is chosen so that it maximizes

$$J(\mathbf{W}) = \frac{|\mathbf{W}^T\mathbf{S}_B\mathbf{W}|}{|\mathbf{W}^T\mathbf{S}_W\mathbf{W}|} \tag{2.18}$$

It can be proved that the matrix $\mathbf{W}$ that maximizes Equation 2.18 is the $m \times k$ matrix that contains the largest eigenvectors of $\mathbf{S}_W^{-1}\mathbf{S}_B$ [5].

Chapter 3

Experiments and results

In this chapter the experiments done to test different dimensionality reduction methods are presented. The data and methods are explained and the results of the experiments performed are analyzed.

3.1 Corpus

To carry on this experiments labelled sentences from the dataset *F1234-m4* were used. That dataset was constructed in the following way: some sentences in Spanish were taken from *Meneame* website (https://www.meneame.net) and given to five different people, that were asked to classify them as sarcastic or not sarcastic. As the sarcasm is sometimes difficult to understand in the written language and can also be ambiguous, the labels that those five people gave to each post did not always coincide. *F1234-m4* dataset contains only the sentences in which at least four out of five people agreed.

The features used to classify the sentences are the number of times that different *ngrams* appear in each sentence. A *ngram* is a group of n words. In this case, unigrams, bigrams and trigrams are counted, i.e., sequences of one word, two words and three words. Moreover, for each sentence the label will be 1 in the case of being sarcastic and -1 if it is not sarcastic. Feature extraction was done through *getFeaturesMNM_stat.py* (see A.1).

There is a total of 2902 posts that generate 230783 features.

Example 3.1.1. Take as an example the following two sentences: 'el verde es el mejor', 'qué rica el agua verde'; where the first one is not sarcastic but the second one is. Then features would be the number of occurrences of the following *ngrams*:

Unigrams: 'el', 'verde', 'es', 'mejor', 'qué', 'rica', 'agua'

Bigrams: 'el verde', 'verde es', 'es el', 'el mejor', 'qué rica', 'rica el', 'el agua', 'agua verde'

Trigrams: 'el verde es', 'verde es el', 'es el mejor', 'qué rica el', 'rica el agua', 'el agua verde'

Then the pattern for the first sentence is (2,1,1,1,0,0,0,1,1,1,1,0,0,0,0,1,1,1,0,0,0) and for the second one (1,1,0,0,1,1,1,0,0,0,0,1,1,1,1,0,0,0,1,1,1). Together with their corresponding labels this two sentences would be represented as ((2,1,1,1,0,0,0,1,1,1,1,0,0,0,0,1,1,1,0,0,0), -1) and ((1,1,0,0,1,1,1,0,0,0,0,1,1,1,1,0,0,0,1,1,1),1).

3.2 *Scikit learn* module

In order to carry out all the experiments done, *Python* programming language has been used, together with the *Scikit learn* module [6]. *Scikit learn* is a free software library that provides machine learning tools in *Python.* Those tools have been used to build the classifier, to do Cross Validation and to apply different dimensionality reduction algorithms. However, one of the methods used, Sequential Forward Selection, was not implemented in *Scikit learn* libraries, so an implementation for it has been designed.

3.3 Multinomial Naive Bayes

The classifier used in this experiments is the Multinomial Naive Bayes classifier. In order to use a Naive Bayes classifier it is necessary to know the distribution of the data. For text classification, it is usual to assume that the data is multinationally distributed, as the features are often counts of words.

This type of classifier was already tested in this same data in a previous final degree dissertation [7]. In that work other classifiers that worked better were also studied, however, Multinomial Naive Bayes was shown to work efficiently and very fast. Thus, and due to the fact that some of the dimensionality reduction methods used are very time consuming, this classifier is a good option as a first approach to test different dimensionality reduction methods.

3.4 Experiments

First of all, classification was done without applying any dimensionality reduction method. This will be used as a base to later check if dimensionality reduction methods improve the performance of the classifier or not. As in all the following experiments, a 10-fold cross validation is performed and the result given here is the average of the results obtained in all the 10 folds. The score used to evaluate the performance of the classification is the F1 score (see Table 1.2). It is important to remember that while using F1 score as an evaluation measure, precision and recall values also need to be checked. Thus, even if those values will not be explicitly written in this work, every time that an F1 score is presented, precision and recall were also analyzed, in order to ensure that they are balanced and avoid false impressions of good performance.

Without introducing any dimensionality reduction method, the average F1 while performing the classification with 10-fold Cross Validation and using Multinomial Naive Bayes classifier is **0.72842**. For the next experiments that will be the reference value to know if dimensionality reduction improves the performance of the classifier or not. Notice that in all the following experiments the conditions (used data, classifier and Cross Validation) are the same. This is a way to ensure that the possible changes on the outcome depend only in the dimensionality reduction method applied each time.

During this work, mostly feature selection methods were tested, as well as one feature transformation method. First of all, some filter type feature selection methods were applied.

3.4.1 Feature selection: filter methods

The simplest one is *Variance Threshold* method, that consist on eliminating all the features with a variance lower than a given value. This method is applied checking the results for different threshold values (see Appendix A.2.5). The threshold value that best results gives is, in most of the 10 folds, around 0.18. However, the average F1 when applying this method is 0.63138, which is lower than the score obtained without dimensionality reduction.

Next, another type of filter methods was applied. In this case, two very similar search procedures will be compared, *SelectKBest* and *SelecPercentile*, which represents the two variations of *Best individual d* explained in Section 2.2.4. Both are ranking methods, that order the features based on the values of a score function and then take the k best features or the $k\%$ best features, respectively. The scores used for ranking are Chi^2 and the ANOVA f-value. Through the experiments described in Appendix A.2.1, Appendix A.2.2, Appendix A.2.3 and Appendix A.2.4 combinations of those two search procedures and score functions were tested, checking for different values of k to see which one gives better results. In the case of *SelectPercentile* k moved from 5 to 95 in steps of 5, while for *SelectKBest* 25 equally spaced values were taken between 1 and the total number of features, to be used as

	SelectKBest	*SelectPercentile*
Chi^2	F1: 0.73159 Average number of features selected: 186467.2 (80.79%)	F1: 0.69358 Average percentage of features selected: 5.5%
f classif	F1: 0.72898 Average number of features selected: 186467.2 (80.79%)	F1: 0.69443 Average percentage of features selected: 5%

Table 3.1: Outcome of the experiments combining *SelectKBest* and *Select-Percentile* search procedures and Chi^2 and f classification score functions. F1 values and the average best performing number or percentage of features are represented.

the value of k. Table 3.1 summarizes the outcomes of those experiments.

As it can be seen in Table 3.1, the results are better when using *SelectKBest*. Using Chi^2 score function baseline result is improved, while f classification maintains baseline value. Whit respect to the search procedure, the results obtained in Table 3.1 show that with *SelectKBest* both trials end up selecting the same number of features, which represents 80.79% of the total features. On the other hand, *SelectPercentile* only maintains around 5% of the total features. Being the two search procedures very similar, it would be logical for them to select a similar amount of features.

In order to clarify the reason for this difference, the results of the previous experiments analyzed fold by fold. In the case of *SelectPercentile*, in average the best results are obtained with 5% of the features when using f classification. But if the results of all the 10 folds are checked, it appears that 5% is not only the average, but the best value in all of the 10 cases. Moreover, in the case of Chi^2, the best result is 5% in all of the cases except of one in which it is 10%. Similarly, while checking fold by fold with *SelectKBest*, with any of the scores it shows that, even if the average is obtained with 80% of the features, in eight out of ten folds the best results were obtained with 99.99% of the features, while the remaining two folds obtain their best at 4% of the features. This suggested that there could be two points in which the F1 value is maximum, one with all the features and one with 4-5% of the features.

To check this hypothesis, more experiments were performed, using both *SelectKBest* and *SelectPercentile* but this time only for f classification score, as both f classification and Chi^2 seemed to give similar results, regarding the number of features to take for best results. The new experiments focused on the areas in which the maximum values have appeared, testing for more

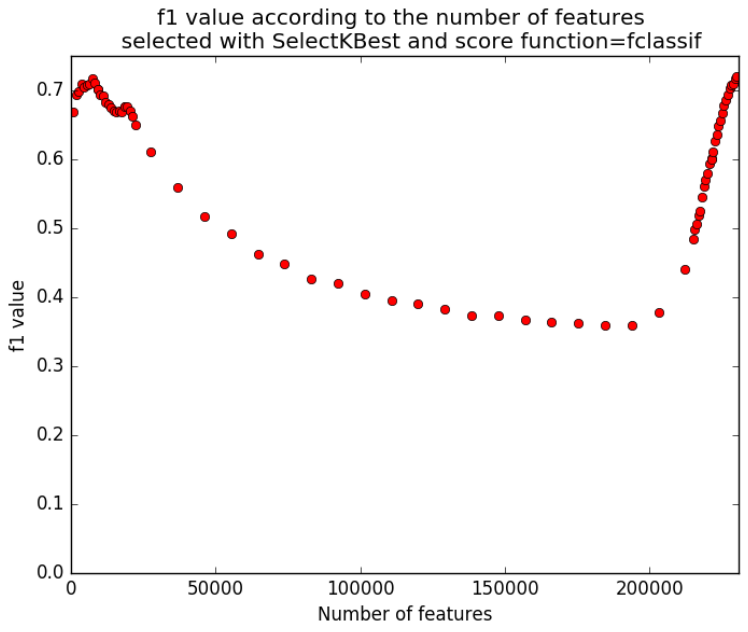

Figure 3.1: F1 score of the classification done as a function of the number of features kept.

values around 5% of features and next to all features. Figure 3.1 shows the results of those experiments with *SelectKBest* search procedure. The results with *SelectPercentile* were similar. While checking only with a small number of features, *SelectKBest* found the best with an average of 7569.6 features (3.28% of the total features) and had an average F1 of 0.72725. *SelectPercentile* had an average F1 of 0.72131 with 3% of the features on average. On the other hand, while checking only with high amounts of features both methods found their best almost at 100% (99.75% and 99.9%) with F1 values of 0.73102 for *SelectKBest* and 0.72964 for *SelectPercentile*. Thus, it can be said that the best results are achieved either with all the features (i.e., without performing feature selection) or with only 3% of the features.

Next step was to check if, while selecting 3% of the features, the features chosen were the same in all the folds. That is, to find out if those 3% of features were an specific set of features that contained the information about the sarcasm or if, on the contrary, they were different from case to case. Focusing only in the case in which a small number of features was maintained, there were 2323 that were kept in all the 10 folds. As said above, on average 7569.6 were preserved, so the common features made an 30.68% of the average kept features. However, the fold that took less features only kept 3693, so it is impossible to have more than 3693 common features. This means that, from all the features that there could be in common in all the folds, 62.9% were indeed in common.

3.4.2 Feature selection: wrapper methods

Regarding wrapper type feature selection methods, two of them were tested: Sequential Backward Selection and Sequential Forward Selection.

Sequential Backward Selection is available in the *Scikit Learn* library and using this method the average number of features kept is 230763.3, which is a 99.99% of the total features. In this way, the average F1 value obtained was 0.73131. Sequential Forward Selection, on the other hand, was not implemented by *Scikit Learn* so an implementation had to be done. There are, in fact, two different results obtained with two different implementations.

The first implementation gave promising results. It obtained a very high average F1 value of 0.95970, selecting on average 75.78 features. This is a very interesting outcome, however, the way in which Sequential Forward Selection was implemented was not completely correct, since it used the test set as validation set. Thus, a second implementation was designed using a different validation set. In this second case, the average number of features is much lower, 34, and the F1 score is only of 0.25205. Even if it was expected to have a lower F1 score in this second implementation (because the model is not specifically adapted to the test set as it was in the previous one), this result is worse than expected. The difference in the number of chosen features between the two suggests that maybe a different variation of the Sequential Forward Selection could be used to obtain a better outcome. That variation stops the process not when the score function decreases, but when it decreases significantly or does not increase in the next iterations. This could be useful if, for example, the F1 value with 34 features is 0.25, with 35 features is 0.24 and with 36 is 0.42.

3.4.3 Feature transformation

Finally, regarding feature transformation, PCA method could not be tested because the machines used were not able to carry out such a complex process for the big amount of data in this project. Instead, LDA was performed, which on average transformed the feature space to one with 230776 dimensions, instead of the original one of 230783 dimensions, and achieved an average F1 of 0.68328.

3.5 Conclusions and future work

The results above presented show that, for this concrete case, dimensionality reduction does not improve significantly the performance of the classifier. Moreover, there are some cases in which applying those methods worsens the performance, such as when features with low variance are eliminated or the case of LDA, which is a very costly method that does not improve performance.

However, even if the performance of the classifier does not improve, it is important to remember that dimensionality reduction has other benefits, such as reducing the complexity of the classifier. Thus, it can be said that applying dimensionality reduction is convenient, specially if the method used is simple. This is the case of reducing features with *SelectKBest* or *SelectPercentile* and any of the two scores used. Those methods are simple, they do not need much time to reduce dimensions and they achieve the same performance with only 3% of the original dimensions, making the classification process much simpler.

As a work for the future, it would be interesting to check the results obtained with the methods that could not be tested, such as PCA or mutual information. Changing the version of Sequential Forward Selection algorithm to avoid stopping too soon could also be done. Finally, another interesting question is to delve into the matter of the common kept features, to see if there are some special words that could be sarcasm indicators.

Bibliography

[1] Richard O. Duda, Peter E. Hart, David G. Stork, *Pattern Classification*, 2nd ed.

[2] Ricardo Gutierrez-Osuna, *CSCE 666: Pattern Analysis* lecture notes, Texas A & M University, 2013.

[3] Andrew R. Webb, Keith D. Cospey, *Statistical Pattern Recognition*, 3rd ed., Wiley, 2011.

[4] M. Inés Torres, *Reconocimiento de formas* lecture notes, 2013.

[5] Ethem Alpaydin, *Introduction to Machine Learning*, MIT Press, 2004.

[6] Scikit-learn.org. (2016). scikit-learn: machine learning in Python scikit-learn 0.17.1 documentation. [online] Available at: http://scikit-learn.org/stable/

[7] Jon Kerexeta, *Sarkasmoaren detekzioa lineako sare sozialetan*, Final Degree Dissertation directed by Raquel Justo and M. Inés Torres, 2016.

Appendix A

Python scripts

A.1 *getFeatures*

```
#!/usr/bin/env python3

# vim:ts=4:sw=4:expandtab:

import re
import csv
import nltk
import json
import pickle
import itertools
import argparse
from glob import glob
from pdb import set_trace
from ast import literal_eval
from operator import itemgetter
from os.path import splitext, split
from nltk.stem.porter import PorterStemmer
from collections import defaultdict, Counter

sarc_cues = set() ## set() -> To create a list with no
    duplicate elements
ngram_files = ['unigrams.csv','bigrams.csv','trigrams.csv']

def getLIWC(text):
    with open('../data/liwc_lookup.json', 'r') as fi:
        liwc_lu = json.load(fi)
    with open('../data/liwc_dict.cPickle', 'rb') as fi:
        liwc = pickle.load(fi)
    results = set()
    for token_tuple in text: ##token=symbol
        token = token_tuple[0]
        for word in liwc_lu:
            if word[-1] == '*':
                if token.startswith(word):
                    for ID in liwc_lu[word]:
                        results.add(('LIWC-'+liwc[ID]['
                            cat_name'],))
            else:
                if token == word:
                    for ID in liwc_lu[word]:
                        results.add(('LIWC-'+liwc[ID]['
                            cat_name'],))
    return list(results)


##def delete_repeated(liwc_feats):
##        for i in range(len(liwc_feats)):
##                j=i+1
##                while j<len(liwc_feats):
##                        if liwc_feats[j]==liwc_feats[i]:
##                                del liwc_feats[j]
##                        else:
##                                j=j+1


def getLength(text_flat, sentences):
    d = Counter()
    d[('LENGTH-total_words',)] = len(text_flat)
    d[('LENGTH-total_char',)] = sum([len(word) for word in
        text_flat])
    d[('LENGTH-total_sentences',)] = len(sentences)
    d[('LENGTH-ave_words_per_sent',)] = len(text_flat) /
        float(len(sentences))
    d[('LENGTH-ave_chars_per_word',)] = d[('LENGTH-
        total_char',)] / float(len(text_flat))
    return d
scue_debug = set()


def getLabel(lpf):
    with open(lpf, encoding='utf-8') as f:
        lpl = json.load(f)

    lpd = {p[0]: dict(label=p[1]) for p in lpl} ## p[0]
        post number, p[1] sarc_si/sarc_no
    return lpd

def getFeatures(lpd):
    q = 0
    header = set()
    for p in lpd:
```

```
with open('C:/Users/SONY/Desktop/Uni/TFG/materiala/
    sofoco/sofoco/Corpus/ParsedPosts/'+p+'.json',
    encoding='utf-8') as f:
     parsed_post = json.load(f)

post_text= [[word.lower() for word in sentence['
    text']] for sentence in parsed_post['sentences'
    ]]

#post_POS = [['POS-'+word[1]['PartOfSpeech'] for
    word in sentence['words']] for sentence in post
    ['sentences']]


# these are nested lists...
#POS_bigrams = [nltk.ngrams(text, 2) for text in
    post_POS]
#POS_trigrams =  [nltk.ngrams(text, 3) for text in
    post_POS]
# so are these
word_unigrams = [nltk.ngrams(text, 1) for text in
    post_text]  #Estaba comentado originalmente
word_bigrams = [nltk.ngrams(text, 2) for text in
    post_text]
word_trigrams = [nltk.ngrams(text, 3) for text in
    post_text]

# flatten lists
#pos_unigrams = list(itertools.chain.from_iterable(
    post_POS))
#pos_bigrams = list(itertools.chain.from_iterable(
    POS_bigrams))
#pos_trigrams = list(itertools.chain.from_iterable(
    POS_trigrams))

#word_unigrams = list(itertools.chain.from_iterable
    (post_text))    #Estaba descomentado
    originalmente
word_unigrams = list(itertools.chain.from_iterable(
    word_unigrams)) #Estaba comentado originalmente
word_bigrams = list(itertools.chain.from_iterable(
    word_bigrams))
word_trigrams = list(itertools.chain.from_iterable(
    word_trigrams))

# get liwc features
#liwc_feats = getLIWC(word_unigrams)
#delete_repeated(liwc_feats)
```

```
        # merge 1 2 3 grams
        #ngrams =  word_unigrams  + word_bigrams +
            word_trigrams + liwc_feats + pos_bigrams +
            word_trigrams + pos_trigrams + pos_unigrams
        #ngrams =  word_unigrams  + word_bigrams +
            word_trigrams + liwc_feats + pos_bigrams +
            pos_trigrams + pos_unigrams
        ngrams =  word_unigrams  + word_bigrams +
            word_trigrams

        # count up all the unique ngrams in this post
        count = Counter()
        for ngram in ngrams:
            header.add(ngram) ## Header is set() so no
                repetitions
            count[ngram] += 1

        #count += getLength(word_unigrams, post_text)

        lpd[p]['ngrams_count'] = count
        q+=1
        # print(q) Estaba sin comentar
    return header

def makeHeaderIndex(header):
    header_index_table = {}
    with open('C:/Users/SONY/Desktop/Uni/TFG/materiala/
        sofoco/sofoco/results/{}/Feature-Index_stat.dat'.
        format(args.posts_id), 'w', encoding='utf-8') as fo:
        for i,h in enumerate(header):
            header_index_table[h] = i # assign unique index
                for each feature
            if isinstance(h, tuple): ## isinistance,
                similar to type
                fo.write("%d:\"\"\"%s\"\"\"\t" % (i, '␣'.
                    join(h)))
            else:
                print('NO_TUPLA,␣SOCORRO:', h)
                fo.write("%d:%s\t"%(i, h))
    return header_index_table


if __name__ == '__main__':

    parser = argparse.ArgumentParser()
    parser.add_argument('posts_id', help='identificador␣del
        ␣conjunto␣de␣posts')
    args = parser.parse_args()
    lbl_posts = getLabel('C:/Users/SONY/Desktop/Uni/TFG/
        materiala/sofoco/sofoco/Corpus/label-{}.json'.format
        (args.posts_id))

    header = getFeatures(lbl_posts)

    #header.add(('LENGTH-total_words',))
    #header.add(('LENGTH-total_char',))
    #header.add(('LENGTH-total_sentences',))
    #header.add(('LENGTH-ave_words_per_sent',))
    #header.add(('LENGTH-ave_chars_per_word',))
    #header.add('SARC_CUE_WORDS')

    header = list(header)
    #for elem in header:
    #     print('header',header)
    header.sort()          # RAQUEL
    #set_trace()
    header.insert(0,'Label')
    header_index_table = makeHeaderIndex(header)

    with open('C:/Users/SONY/Desktop/Uni/TFG/materiala/
        sofoco/sofoco/results/{}/results_stat.txt'.format(
        args.posts_id), 'w') as f:
         for p in sorted(lbl_posts):
             if lbl_posts[p]['label'] == 'sarc_si':
                 f.write('1\t')
             else:
                 f.write('-1\t')

##              try:
##                  assert p in lbl_posts
##              except:
##                  # These files have no text/POS or
    something else bad happened
##                  print("Warning: Bad parse in file: %s.
    json"%str(k))

             tuple_pairs = []
             elems=[]
             for ngram in lbl_posts[p]['ngrams_count']:
                 i = header_index_table[ngram]
                 tuple_pairs.append((i,lbl_posts[p]['
                     ngrams_count'][ngram]))
             tuple_pairs.sort() # svm-light only takes
                 features with increasing  index values
             for pair in tuple_pairs:

                 #f.write(str(pair[0])+":"+str(pair[1])+"
                     ")
                 #print("pair",pair)

                 f.write("%d:%d\t\t"%pair)

             f.write('\n')
    print("Total number of features: %d"%len(header))
    print("Total number of posts: %d"%len(lbl_posts))
```

A.2 Experiments

A.2.1 Chi^2

```
#!/usr/bin/env python3

# vim: ts=4:sw=4:expandtab:

from pdb import runcall, set_trace
import sklearn.datasets.svmlight_format as svmlight
from sklearn.feature_selection import chi2, SelectKBest
from sklearn.naive_bayes import MultinomialNB
from sklearn import svm
from sklearn.metrics import f1_score, precision_score,
    recall_score, accuracy_score
from sklearn.cross_validation import KFold, StratifiedKFold
from operator import itemgetter
from random import shuffle
from numpy import array
import argparse
import sys


def fitAndPredict(trainI, testI, k, feat_vecs, labels):
    """ do one round of fitting to chi2, making model, and
        doing predictions
    on test"""
    ch2 = SelectKBest(chi2, k=k)
    best = ch2.fit(feat_vecs[trainI], labels[trainI]) #do
        chi2 fit on train data
    #best = runcall(ch2.fit, feat_vecs[trainI], labels[
        trainI]) #do chi2 fit on train data
    test_feats = best.transform(feat_vecs[testI])    # test
        data reduced to same k features
    train_feats = best.transform(feat_vecs[trainI]) # train
         data reduced to same k features
    train_labels = labels[trainI] # labels for this sample
        section

    fitted = clf.fit(train_feats, train_labels) # make NB
```

```
        model on train data ##clf is the classifier and we
        fit it to our train data
    pred = fitted.predict(test_feats) # predict labels for
        test
    return pred

def iterK(max_k, step_size, trainI, testI, i, feat_vecs, labels,
    test_labels):
    """ Search through all k, return prediction of model
        with
    higest F1"""
    f1_k = []
    print("fold ---------------> " + str(i))
    for k in range(1,max_k,step_size):
        pred = fitAndPredict(trainI, testI, k, feat_vecs,
            labels)

        if k != 1:
            f1_k.append((f1_score(test_labels, pred), k))
            print("%d features: f1=%.4f"%(f1_k[-1][1], f1_k
                [-1][0]))
    k = sorted(f1_k, key=itemgetter(0))[-1][1] # take
        highest F1 ##sort f1_k but te key is sort it by the
        0. item, that is the f1 score.
                ## Then take the k that gives the highest
                    F1, the last k
    pred = fitAndPredict(trainI, testI, k, feat_vecs, labels) #
        # We are going to use that k to do pred
    # compute scores
    k_list.append(k)
    return pred


def xval(svm_light_in, num_folds, num_steps=25, get_best=True,
    k=None):

    global k_list

    feat_vecs, labels = svmlight.load_svmlight_file(
        svm_light_in)
    ## feat_vecs is a sparse mat that has rows:
    ##("n  of post", "n  (identifier) of ngram") "n  of
        times that ngram is in that post"
    ## labels is a list that contains 1s and -1s (for sarc/
        no sarc)

    max_k=feat_vecs.shape[1] ## max_k= number of ngrams (at
         most I can take max_k features)
    print('MAX_K: ', max_k)
```

```
step_size = max_k//num_steps

kf = StratifiedKFold(labels, n_folds=num_folds, shuffle
    =False) # make folds
#kf = StratifiedKFold(labels, n_folds=num_folds,
    shuffle=True) # make folds

f1 = []
accuracy = []
precision = []
recall = []
k_list = []
i = 1
for trainI,testI in kf: ## for each fold (the folds don
    't have the same size)
     test_labels = labels[testI]
     if get_best: # search though k's for best results
        ## This is my case if I want to do the feature
        selection
         pred = iterK(max_k,step_size,trainI,testI,i,
             feat_vecs,labels,test_labels)
     else: ## just do one model for k='all', without FS
         pred = fitAndPredict(trainI,testI,k,feat_vecs,
             labels)


     f1.append(f1_score(test_labels,pred))
     accuracy.append(accuracy_score(test_labels,pred))
     precision.append(precision_score(test_labels,pred))
     recall.append(recall_score(test_labels,pred))
     i+=1
f1 = array(f1) ## converts f1 list into numpy.ndarray
    type
accuracy = array(accuracy)
precision = array(precision)
recall = array(recall)
k_list = array(k_list)
print('Avg␣F1:␣' + str(f1.mean()))
print('Avg␣Accuracy:␣' + str(accuracy.mean()))
print('Avg␣Precision:␣' + str(precision.mean()))
print('Avg␣Recall:␣' + str(recall.mean() ))
if get_best: print('Avg␣k:␣' + str(k_list.mean()))
print(f1)
print(accuracy)
print(precision)
print(recall)
print(k_list)
return [score.mean() for score in (f1,accuracy,
    precision,recall)]
```

```
parser = argparse.ArgumentParser()
#parser.add_argument('posts_id', help='identificador del
    conjunto de posts')
#parser.add_argument('nu', help='factor nu para NuSVC (
    intervalo (0,1])', type=float, default=0.6)
parser.add_argument('-a', '--archivo', help='guardar salida
    en archivo', action='store_true')
parser.add_argument('--folds', help='n mero de folds [10]'
    , type=int, default=10)
args = parser.parse_args()

if args.archivo:
    sys.stdout = open('stat-MNB-SVM-withFS_CHI2.txt', 'w',
        buffering=1)

print("Loading dataset ...")
svm_light_in = 'results_stat.txt'
## stvm_light_in stores the name of the results_stats
    document
## in results_stat : for each post, 1 if sarc/-1 if not
    sarc and the number(identifier) of the ngram : how many
    times apears that ngram in that post

# initialize classifier
clf = MultinomialNB()
#clf = svm.NuSVC(nu=args.nu)

# xval(svm_light_in, args.folds) # regular Xval
xval(svm_light_in, args.folds, get_best=True, k='all')
```

A.2.2 PercentileChi^2

```
#!/usr/bin/env python3

# vim:ts=4:sw=4:expandtab:

from pdb import runcall, set_trace
import sklearn.datasets.svmlight_format as svmlight
from sklearn.feature_selection import chi2, SelectKBest
from sklearn.naive_bayes import MultinomialNB
from sklearn import svm
from sklearn.metrics import f1_score, precision_score,
    recall_score, accuracy_score
from sklearn.cross_validation import KFold, StratifiedKFold
from operator import itemgetter
from random import shuffle
from numpy import array
```

```
import argparse
import sys
from sklearn.feature_selection import SelectPercentile


def fitAndPredict(trainI, testI, k, feat_vecs, labels):
    """ do one round of fitting to chi2, making model, and
        doing predictions
    on test"""
    ch2 = SelectPercentile(chi2, k) ## only keeps the best
        k% features according to chi2
    best = ch2.fit(feat_vecs[trainI], labels[trainI]) #do
        chi2 fit on train data
    test_feats = best.transform(feat_vecs[testI])   # test
        data reduced to same k features
    train_feats = best.transform(feat_vecs[trainI]) # train
         data reduced to same k features
    train_labels = labels[trainI] # labels for this sample
        section

    fitted = clf.fit(train_feats, train_labels) # make NB
        model on train data
    pred = fitted.predict(test_feats) # predict labels for
        test
    return pred

def iterK(max_k, step_size, trainI, testI, i, feat_vecs, labels,
    test_labels):
    """ Search through all k, return prediction of model
        with
    higest F1"""
    f1_k = []
    print("fold------------------> " + str(i))
    k=95
    while k>=5:
        pred = fitAndPredict(trainI, testI, k, feat_vecs,
            labels)
        f1_k.append((f1_score(test_labels, pred), k))
        print("%d percent of features: f1=%.4f"%(f1_k
            [-1][1], f1_k[-1][0]))
        k=k-5
    k = sorted(f1_k, key=itemgetter(0))[-1][1]
    pred = fitAndPredict(trainI, testI, k, feat_vecs, labels)
    # compute scores
    k_list.append(k)
    return pred


def xval(svm_light_in, num_folds, num_steps=25, get_best=True,
   k=None):

    global k_list

    feat_vecs, labels = svmlight.load_svmlight_file(
        svm_light_in)

    max_k=feat_vecs.shape[1]
    print('MAX_K: ', max_k)
    step_size = max_k//num_steps

    kf = StratifiedKFold(labels, n_folds=num_folds, shuffle
        =False) # make folds

    f1 = []
    accuracy = []
    precision = []
    recall = []
    k_list = []
    i = 1
    for trainI, testI in kf:
        test_labels = labels[testI]
        if get_best: # search though k's for best results
            pred = iterK(max_k, step_size, trainI, testI, i,
                feat_vecs, labels, test_labels)
        else: # just do one model for k
            pred = fitAndPredict(trainI, testI, k, feat_vecs,
                labels)


        f1.append(f1_score(test_labels, pred))
        accuracy.append(accuracy_score(test_labels, pred))
        precision.append(precision_score(test_labels, pred))
        recall.append(recall_score(test_labels, pred))
        i+=1
    f1 = array(f1)
    accuracy = array(accuracy)
    precision = array(precision)
    recall = array(recall)
    k_list = array(k_list)
    print('Avg F1: ' + str(f1.mean()))
    print('Avg Accuracy: ' + str(accuracy.mean()))
    print('Avg Precision: ' + str(precision.mean()))
    print('Avg Recall: ' + str(recall.mean() ))
    if get_best: print('Avg k: ' + str(k_list.mean()))
    print(f1)
    print(accuracy)
    print(precision)
    print(recall)
```

```
    print(k_list)
    return [score.mean() for score in (f1,accuracy,
        precision,recall)]


parser = argparse.ArgumentParser()
#parser.add_argument('posts_id', help='identificador del
    conjunto de posts')
#parser.add_argument('nu', help='factor nu para NuSVC (
    intervalo (0,1])', type=float, default=0.6)
parser.add_argument('-a', '--archivo', help='guardar␣salida
    ␣en␣archivo', action='store_true')
parser.add_argument('--folds', help='n mero␣de␣folds␣[10]'
    , type=int, default=10)
args = parser.parse_args()

if args.archivo:
    sys.stdout = open('stat-MNB-SVM-withFS_percentile_CHI2.
        txt', 'w', buffering=1)

print("Loading␣dataset␣...")
svm_light_in = 'results_stat.txt'

# initialize classifier
clf = MultinomialNB()

xval(svm_light_in,args.folds,get_best=True,k='all')
```

A.2.3 fclassif

```
#!/usr/bin/env python3

# vim:ts=4:sw=4:expandtab:

from pdb import runcall,set_trace
import sklearn.datasets.svmlight_format as svmlight
from sklearn.feature_selection import f_classif,
    SelectKBest
from sklearn.naive_bayes import MultinomialNB
from sklearn import svm
from sklearn.metrics import f1_score,precision_score,
    recall_score, accuracy_score
from sklearn.cross_validation import KFold,StratifiedKFold
from operator import itemgetter
from random import shuffle
from numpy import array, intersect1d
import argparse
import sys
from functools import reduce
```

```
def fitAndPredict(trainI, testI, k, feat_vecs, labels):
    """ do one round of fitting to chi2, making model, and
        doing predictions
    on test"""
    fclass = SelectKBest(f_classif, k=k)
    best = fclass.fit(feat_vecs[trainI].toarray(), labels[
        trainI]) #do chi2 fit on train data
    test_feats = best.transform(feat_vecs[testI])    # test
        data reduced to same k features
    train_feats = best.transform(feat_vecs[trainI]) # train
         data reduced to same k features
    train_labels = labels[trainI] # labels for this sample
        section

    indices=best.get_support()

    fitted = clf.fit(train_feats, train_labels) # make NB
        model on train data
    pred = fitted.predict(test_feats) # predict labels for
        test
    return pred, indices

def iterK(max_k, step_size, trainI, testI, i, feat_vecs, labels,
    test_labels):
    """ Search through all k, return prediction of model
        with
    higest F1"""
    f1_k = []
    print("fold---------------> " + str(i))
    for k in range(1, max_k, step_size):
        pred, indices= fitAndPredict(trainI, testI, k,
            feat_vecs, labels) #i dont need indices right now
             but if i only put pred it stores both pred and
            indices
        if k != 1:
            f1_k.append((f1_score(test_labels, pred), k))
            print("%d features: f1=%.4f"%(f1_k[-1][1], f1_k
                [-1][0]))
    k = sorted(f1_k, key=itemgetter(0))[-1][1]
    pred, goodindices = fitAndPredict(trainI, testI, k,
        feat_vecs, labels)
    # compute scores
    k_list.append(k)
    return pred, goodindices


def xval(svm_light_in, num_folds, num_steps=25, get_best=True,
   k=None):

    global k_list

    feat_vecs,labels = svmlight.load_svmlight_file(
        svm_light_in)

    max_k=feat_vecs.shape[1]
    print('MAX_K: ', max_k)
    step_size = max_k//num_steps

    kf = StratifiedKFold(labels, n_folds=num_folds, shuffle
        =False) # make folds

    f1 = []
    accuracy = []
    precision = []
    recall = []
    k_list = []
    i = 1
    for trainI,testI in kf:
        test_labels = labels[testI]
        if get_best:
            pred, indices = iterK(max_k,step_size,trainI,
                testI,i,feat_vecs,labels,test_labels)
            print('indices')
            print(indices)
            if i==1:
                shared_indices=indices
            else:
                shared_indices=intersect1d(shared_indices,
                    indices)
        else: # just do one model for k
            pred = fitAndPredict(trainI,testI,k,feat_vecs,
                labels)

        f1.append(f1_score(test_labels,pred))
        accuracy.append(accuracy_score(test_labels,pred))
        precision.append(precision_score(test_labels,pred))
        recall.append(recall_score(test_labels,pred))
        i+=1
    f1 = array(f1)
    accuracy = array(accuracy)
    precision = array(precision)
    recall = array(recall)
    k_list = array(k_list)
    print('Avg F1: ' + str(f1.mean()))
    print('Avg Accuracy: ' + str(accuracy.mean()))
    print('Avg Precision: ' + str(precision.mean()))
```

```
    print('Avg Recall: ' + str(recall.mean() ))
    if get_best: print('Avg k: ' + str(k_list.mean()))
    print(f1)
    print(accuracy)
    print(precision)
    print(recall)
    print(k_list)
    print('shared_indices', shared_indices)
    print('length: ', len(shared_indices))
    for i in shared_indices:
        print(i)
    return [score.mean() for score in (f1,accuracy,
        precision,recall)]


parser = argparse.ArgumentParser()
#parser.add_argument('posts_id', help='identificador del
    conjunto de posts')
#parser.add_argument('nu', help='factor nu para NuSVC (
    intervalo (0,1])', type=float, default=0.6)
parser.add_argument('-a', '--archivo', help='guardar salida
    en archivo', action='store_true')
parser.add_argument('--folds', help='n mero de folds [10]'
    , type=int, default=10)
args = parser.parse_args()

if args.archivo:
    sys.stdout = open('stat-MNB-SVM-withFS_fclassif_indices
        .txt', 'w', buffering=1)

print("Loading dataset ...")
svm_light_in = 'results_stat.txt'

clf = MultinomialNB()

xval(svm_light_in,args.folds,get_best=True,k='all')
```

A.2.4 Percentile fclassif

```
#!/usr/bin/env python3

# vim:ts=4:sw=4:expandtab:

from pdb import runcall,set_trace
import sklearn.datasets.svmlight_format as svmlight
from sklearn.feature_selection import f_classif,
    SelectKBest
from sklearn.naive_bayes import MultinomialNB
from sklearn import svm
from sklearn.metrics import f1_score,precision_score,
    recall_score, accuracy_score
from sklearn.cross_validation import KFold,StratifiedKFold
from operator import itemgetter
from random import shuffle
from numpy import array
import argparse
import sys
from sklearn.feature_selection import SelectPercentile

def fitAndPredict(trainI,testI,k,feat_vecs,labels):
    """ do one round of fitting to chi2, making model, and
        doing predictions
    on test"""
    fclassif = SelectPercentile(f_classif, k)
    best = fclassif.fit(feat_vecs[trainI].toarray(), labels
        [trainI])
    test_feats = best.transform(feat_vecs[testI])
    train_feats = best.transform(feat_vecs[trainI])
    train_labels = labels[trainI] # labels for this sample
        section
    print('train␣feats', train_feats.shape)

    fitted = clf.fit(train_feats, train_labels) # make NB
        model on train data
    pred = fitted.predict(test_feats) # predict labels for
        test
    return pred

def iterK(max_k,step_size,trainI,testI,i,feat_vecs,labels,
    test_labels):
    """ Search through all k, return prediction of model
        with
    higest F1"""
    f1_k = []
    print("fold-------------->␣" + str(i))
    k=95
    while k>=5:
        pred = fitAndPredict(trainI,testI,k,feat_vecs,
            labels)
        f1_k.append((f1_score(test_labels, pred), k))
        print("%d␣percent␣of␣features:␣f1=%.4f"%(f1_k
            [-1][1], f1_k[-1][0]))
        k=k-5
    k = sorted(f1_k, key=itemgetter(0))[-1][1]
    pred = fitAndPredict(trainI,testI,k,feat_vecs,labels)
    # compute scores
    k_list.append(k)
    return pred
```

```
def xval(svm_light_in,num_folds,num_steps=25,get_best=True,
    k=None):

    global k_list

    feat_vecs,labels = svmlight.load_svmlight_file(
        svm_light_in)

    max_k=feat_vecs.shape[1]
    print('MAX_K: ', max_k)
    step_size = max_k//num_steps

    kf = StratifiedKFold(labels, n_folds=num_folds, shuffle
        =False) # make folds

    f1 = []
    accuracy = []
    precision = []
    recall = []
    k_list = []
    i = 1
    for trainI,testI in kf:
        test_labels = labels[testI]
        if get_best:
            pred = iterK(max_k,step_size,trainI,testI,i,
                feat_vecs,labels,test_labels)
        else:
            pred = fitAndPredict(trainI,testI,k,feat_vecs,
                labels)

        f1.append(f1_score(test_labels,pred))
        accuracy.append(accuracy_score(test_labels,pred))
        precision.append(precision_score(test_labels,pred))
        recall.append(recall_score(test_labels,pred))
        i+=1
    f1 = array(f1)
    accuracy = array(accuracy)
    precision = array(precision)
    recall = array(recall)
    k_list = array(k_list)
    print('Avg␣F1:␣' + str(f1.mean()))
    print('Avg␣Accuracy:␣' + str(accuracy.mean()))
    print('Avg␣Precision:␣' + str(precision.mean()))
    print('Avg␣Recall:␣' + str(recall.mean() ))
    if get_best: print('Avg␣k:␣' + str(k_list.mean()))
    print(f1)
    print(accuracy)
    print(precision)
    print(recall)
    print(k_list)
    return [score.mean() for score in (f1,accuracy,
        precision,recall)]


parser = argparse.ArgumentParser()
#parser.add_argument('posts_id', help='identificador del
    conjunto de posts')
#parser.add_argument('nu', help='factor nu para NuSVC (
    intervalo (0,1])', type=float, default=0.6)
parser.add_argument('-a', '--archivo', help='guardar␣salida
    ␣en␣archivo', action='store_true')
parser.add_argument('--folds', help='n mero␣de␣folds␣[10]'
    , type=int, default=10)
args = parser.parse_args()

if args.archivo:
    sys.stdout = open('stat-MNB-SVM-
        withFS_percentile_fclassif.txt', 'w', buffering=1)

print("Loading␣dataset␣...")
svm_light_in = 'results_stat.txt'

clf = MultinomialNB()

xval(svm_light_in,args.folds,get_best=True,k='all') ## I've
     changed get_best from false to true to do feature
    selection
```

A.2.5 Variance Threshold

```
#!/usr/bin/env python3

# vim:ts=4:sw=4:expandtab:


from pdb import runcall,set_trace
import sklearn.datasets.svmlight_format as svmlight
from sklearn.feature_selection import chi2, SelectKBest
from sklearn.naive_bayes import MultinomialNB
from sklearn import svm
from sklearn.metrics import f1_score,precision_score,
    recall_score, accuracy_score
from sklearn.cross_validation import KFold,StratifiedKFold
from operator import itemgetter
from random import shuffle
from numpy import array
```

```
import argparse
import sys
from sklearn.feature_selection import VarianceThreshold
from sklearn import preprocessing
from numpy import std
from numpy import arange

def iterK(max_k,step_size,trainI,testI,i,feat_vecs,labels,
    test_labels):
     """ Search through all k, return prediction of model
         with
     higest F1"""
     f1_k = []
     print("fold--------------------> " + str(i))
     for k in arange(0,max_k,step_size): #for different
         values for the variance

          sel=VarianceThreshold(threshold=k)
          best=sel.fit(feat_vecs[trainI],labels[trainI])
          test_feats=best.transform(feat_vecs[testI])
          train_feats=best.transform(feat_vecs[trainI])
          train_labels=labels[trainI]

          fitted=clf.fit(train_feats, train_labels)
          pred = fitted.predict(test_feats)

          f1_k.append((f1_score(test_labels, pred), k))
          print("taking out features with variance %f : f1
              =%.4f"%(f1_k[-1][1], f1_k[-1][0]))

     k = sorted(f1_k, key=itemgetter(0))[-1][1]

     sel=VarianceThreshold(threshold=k)
     best=sel.fit(feat_vecs[trainI], labels[trainI])
     test_feats=best.transform(feat_vecs[testI])
     train_feats=best.transform(feat_vecs[trainI])
     train_labels=labels[trainI]

     fitted= clf.fit(train_feats, train_labels)
     pred= fitted.predict(test_feats)

     k_list.append(k)
     return pred


def xval(svm_light_in,num_folds,num_steps=25,get_best=True,
    k=None):

     global k_list
```

```
feat_vecs, labels = svmlight.load_svmlight_file(
    svm_light_in)

max_k=max(std(feat_vecs.toarray(), axis=0)) #I measure
    the variance of my features and take as max_k the
    maximum
print('MAX_K: ', max_k)
step_size = max_k/num_steps

kf = StratifiedKFold(labels, n_folds=num_folds, shuffle
    =False) # make folds

f1 = []
accuracy = []
precision = []
recall = []
k_list = []
i = 1

for trainI, testI in kf:
     test_labels = labels[testI]
     if get_best:
         pred = iterK(max_k, step_size, trainI, testI, i,
             feat_vecs, labels, test_labels)
     else:
           #pred = fitAndPredict(trainI, testI, k,
             feat_vecs, labels)

     f1.append(f1_score(test_labels, pred))
     accuracy.append(accuracy_score(test_labels, pred))
     precision.append(precision_score(test_labels, pred))
     recall.append(recall_score(test_labels, pred))
     i+=1
f1 = array(f1)
accuracy = array(accuracy)
precision = array(precision)
recall = array(recall)
k_list = array(k_list)
print('Avg F1: ' + str(f1.mean()))
print('Avg Accuracy: ' + str(accuracy.mean()))
print('Avg Precision: ' + str(precision.mean()))
print('Avg Recall: ' + str(recall.mean() ))
if get_best: print('Avg k: ' + str(k_list.mean()))
print(f1)
print(accuracy)
print(precision)
print(recall)
print(k_list)
```

```
    print()
    return [score.mean() for score in (f1,accuracy,
        precision,recall)]

parser = argparse.ArgumentParser()
#parser.add_argument('posts_id', help='identificador del
    conjunto de posts')
#parser.add_argument('nu', help='factor nu para NuSVC (
    intervalo (0,1])', type=float, default=0.6)
parser.add_argument('-a', '--archivo', help='guardar␣salida
    ␣en␣archivo', action='store_true')
parser.add_argument('--folds', help='n mero␣de␣folds␣[10]'
    , type=int, default=10)
args = parser.parse_args()

if args.archivo:
    sys.stdout = open('stat-MNB-SVM-
        withFS_removelowvar_CHI2.txt', 'w', buffering=1)

print("Loading␣dataset␣...")
svm_light_in = 'results_stat.txt'

clf = MultinomialNB()

xval(svm_light_in,args.folds,get_best=True,k='all')
```

A.2.6 Sequential Backward Selection

```
#!/usr/bin/env python3

# vim:ts=4:sw=4:expandtab:


from pdb import runcall,set_trace
import sklearn.datasets.svmlight_format as svmlight
from sklearn.feature_selection import chi2, SelectKBest
from sklearn.naive_bayes import MultinomialNB
from sklearn import svm
from sklearn.metrics import f1_score,precision_score,
    recall_score, accuracy_score
from sklearn.cross_validation import KFold,StratifiedKFold
from operator import itemgetter
from random import shuffle
from numpy import array
import argparse
import sys
from sklearn.feature_selection import RFECV
```

```
def SBS(max_k, step_size, trainI, testI, i, feat_vecs, labels,
    test_labels):
    """ Sequential backward selction"""
    print("fold ——————————> " + str(i))
    estimator= MultinomialNB()
    selector=RFECV(estimator)

    best=selector.fit(feat_vecs[trainI], labels[trainI])
    test_feats=best.transform(feat_vecs[testI])
    train_feats=best.transform(feat_vecs[trainI])
    train_labels=labels[trainI]

    fitted = clf.fit(train_feats, train_labels) # make NB
        model on train data
    pred = fitted.predict(test_feats) # predict labels for
        testI

    return pred

def xval(svm_light_in, num_folds, num_steps=25, get_best=True,
    k=None):

    global k_list

    feat_vecs, labels = svmlight.load_svmlight_file(
        svm_light_in)

    max_k=feat_vecs.shape[1]
    print('MAX_K: ', max_k)
    step_size = max_k//num_steps

    kf = StratifiedKFold(labels, n_folds=num_folds, shuffle
        =False) # make folds

    f1 = []
    accuracy = []
    precision = []
    recall = []
    k_list = []
    i = 1
    for trainI, testI in kf:
        test_labels = labels[testI]
        if get_best:
            pred = SBS(max_k, step_size, trainI, testI, i,
                feat_vecs, labels, test_labels)
        else: # just do one model for k
            #pred = fitAndPredict(trainI, testI, k, feat_vecs,
                labels)
            f1.append(f1_score(test_labels,pred))
            accuracy.append(accuracy_score(test_labels,pred))
            precision.append(precision_score(test_labels,pred))
            recall.append(recall_score(test_labels,pred))
            i+=1
    f1 = array(f1)
    accuracy = array(accuracy)
    precision = array(precision)
    recall = array(recall)
    k_list = array(k_list)
    print('Avg F1: ' + str(f1.mean()))
    print('Avg Accuracy: ' + str(accuracy.mean()))
    print('Avg Precision: ' + str(precision.mean()))
    print('Avg Recall: ' + str(recall.mean() ))
    if get_best: print('Avg k: ' + str(k_list.mean()))
    print(f1)
    print(accuracy)
    print(precision)
    print(recall)
    print(k_list)
    return [score.mean() for score in (f1,accuracy,
        precision,recall)]


parser = argparse.ArgumentParser()
#parser.add_argument('posts_id', help='identificador del
    conjunto de posts')
#parser.add_argument('nu', help='factor nu para NuSVC (
    intervalo (0,1])', type=float, default=0.6)
parser.add_argument('-a', '--archivo', help='guardar salida
    en archivo', action='store_true')
parser.add_argument('--folds', help='n mero de folds [10]'
    , type=int, default=10)
args = parser.parse_args()

if args.archivo:
    sys.stdout = open('stat-MNB-SVM-withFS_SBS_CHI2.txt', '
        w', buffering=1)

print("Loading dataset ...")
svm_light_in = 'results_stat.txt'

clf = MultinomialNB()

xval(svm_light_in,args.folds,get_best=True,k='all')
```

A.2.7 Sequential Forward Selection

```
#!/usr/bin/env python3
```

```
# vim:ts=4:sw=4:expandtab:

from pdb import runcall,set_trace
import sklearn.datasets.svmlight_format as svmlight
from sklearn.feature_selection import chi2, SelectKBest,
    f_regression
from sklearn.naive_bayes import MultinomialNB
from sklearn import svm
from sklearn.metrics import f1_score,precision_score,
    recall_score, accuracy_score
from sklearn.cross_validation import KFold,StratifiedKFold
from operator import itemgetter
from random import shuffle
from numpy import array
import argparse
import sys
from scipy.sparse import *
from scipy import *
from scipy.sparse import hstack

def get_f1(ttrain_feats,ttrain_labels, ttest_feats,
    ttest_labels):
    fitted = clf.fit(ttrain_feats, ttrain_labels)
    pred = fitted.predict(ttest_feats)
    f1=f1_score(ttest_labels,pred)
    return f1

def SFS(feat_vecs, trainI, testI, labels,num_folds,
    print_steps=True):
    """
    Implementation of a Sequential Forward Selection
        algorithm.
    Finds the feature subset using SFS and returns the
        prediciton dor the test post with those features and
         the number of features used.
    """

    # Initialization
    best_f1=0
    good_f1=0

    kf = StratifiedKFold(labels[trainI], n_folds=num_folds,
         shuffle=False)
    #Take train and use one part for train and one for
        validation
    for train, test in kf:
        ttrainI=train
        ttestI=test
    #Only use one of the partition, the last in the 'for'
        in this case

    ttrain_labels=labels[ttrainI]
    ttest_labels=labels[ttestI]

    test_feats=feat_vecs[testI]

    feat_number=feat_vecs.shape[1]

    already_used=[] # In this list the indeces of the
        columns selcected, not to check them again

    #for the first:
    candidates_and_f1=[] ## Empty list that later will be
        filled with tuples (f1 score, candidate, i)
    for i in range(feat_number):
        candidate=feat_vecs.getcol(i)
        ttest_feats=candidate[ttestI]
        ttrain_feats=candidate[ttrainI]
        candidate_f1=get_f1(ttrain_feats, ttrain_labels,
            ttest_feats, ttest_labels)
        candidates_and_f1.append((candidate_f1,candidate,i)
            )
    candidates_and_f1=sorted(candidates_and_f1, key=
        itemgetter(0)) ##order them by f1
    good_f1, good_candidate,good_i=candidates_and_f1.pop()
        #take the las element (the one with biggest f1)

    if good_f1 >= best_f1: # >= because it can be 0
        best_f1=good_f1
        feat_subs= good_candidate
        already_used.append(good_i)

    #for the rest
    I_am_improving=True
    while I_am_improving and len(already_used) <
        feat_number: # while there is something to look at
        and results improved in the last iteration
        candidates_and_f1=[]
        for i in range(feat_number):
            if i not in already_used:
                candidate=feat_vecs.getcol(i)
                ttest_feats=hstack([feat_subs, candidate],
                    format="csr")[ttestI] #Take the already
                    selected and the candidate feature in
                    all the test posts
                ttrain_feats=hstack([feat_subs, candidate],
                    format="csr")[ttrainI]
                candidate_f1=get_f1(ttrain_feats,
                    ttrain_labels, ttest_feats, ttest_labels
                    )
                candidates_and_f1.append((candidate_f1,
                    candidate,i))
        candidates_and_f1=sorted(candidates_and_f1, key=
            itemgetter(0)) ##order them by f1
        good_f1, good_candidate,good_i=candidates_and_f1.
            pop() #take the las element (the one with
            biggest f1)
        if good_f1 > best_f1:
            best_f1=good_f1
            feat_subs= hstack([feat_subs, good_candidate])
            already_used.append(good_i)
            if print_steps:
                print('With␣', str(feat_subs.shape[1]), '
                    features,␣the␣f1␣for␣the␣validation␣set␣
                    is:␣', best_f1)
        else:
            I_am_improving=False

    train_feat_subs=feat_subs.tocsr()[trainI]
    train_labels=labels[trainI]
    test_feats=feat_subs.tocsr()[testI]
    fitted = clf.fit(train_feat_subs, train_labels)
    pred = fitted.predict(test_feats) #try in the test set

    n_features_used=feat_subs.shape[1]

    return pred, n_features_used



def xval(svm_light_in,num_folds,num_steps=25,get_best=True,
    k=None):

    global k_list

    feat_vecs,labels = svmlight.load_svmlight_file(
        svm_light_in)

    max_k=feat_vecs.shape[1]
    print('MAX_K: ', max_k)
    step_size = max_k//num_steps

    kf = StratifiedKFold(labels, n_folds=num_folds, shuffle
        =False)

    f1 = []
    accuracy = []
    precision = []
    recall = []
    n_list = []
    i = 1
    for trainI,testI in kf:
        print('fold␣----------->␣', str(i))
        if get_best:
            pred, n_features = SFS(feat_vecs, trainI, testI
                , labels,num_folds, print_steps=True)
            n_list.append(n_features)
        else:
            #pred = fitAndPredict(trainI,testI,k,feat_vecs,
                labels)

        test_labels=labels[testI]
        f1.append(f1_score(test_labels,pred))
        accuracy.append(accuracy_score(test_labels,pred))
        precision.append(precision_score(test_labels,pred))
        recall.append(recall_score(test_labels,pred))
        i+=1
    f1 = array(f1)
    accuracy = array(accuracy)
    precision = array(precision)
    recall = array(recall)
    n_list = array(n_list)
    print('Avg␣F1:␣' + str(f1.mean()))
    print('Avg␣Accuracy:␣' + str(accuracy.mean()))
    print('Avg␣Precision:␣' + str(precision.mean()))
    print('Avg␣Recall:␣' + str(recall.mean() ))
    if get_best: print('Avg␣n:␣' + str(n_list.mean()))
    print(f1)
    print(accuracy)
    print(precision)
    print(recall)
    print(n_list)
    return [score.mean() for score in (f1,accuracy,
        precision,recall)]


parser = argparse.ArgumentParser()
#parser.add_argument('posts_id', help='identificador del
    conjunto de posts')
#parser.add_argument('nu', help='factor nu para NuSVC (
    intervalo (0,1])', type=float, default=0.6)
parser.add_argument('-a', '--archivo', help='guardar␣salida
    ␣en␣archivo', action='store_true')
parser.add_argument('--folds', help='n mero␣de␣folds␣[10]'
    , type=int, default=10)
```

```
args = parser.parse_args()

if args.archivo:
    sys.stdout = open('stat-MNB-SVM-withFS_SFS_welldone.txt
        ', 'w', buffering=1)

print("Loading␣dataset␣...")
svm_light_in = 'results_stat.txt'

clf = MultinomialNB()

xval(svm_light_in, args.folds, get_best=True, k='all')
```

A.2.8 LDA

```
#!/usr/bin/env python3

# vim:ts=4:sw=4:expandtab:


from pdb import runcall, set_trace
import sklearn.datasets.svmlight_format as svmlight
from sklearn.feature_selection import f_classif,
    SelectKBest
from sklearn.naive_bayes import MultinomialNB
from sklearn import svm
from sklearn.metrics import f1_score, precision_score,
    recall_score, accuracy_score
from sklearn.cross_validation import KFold, StratifiedKFold
from operator import itemgetter
from random import shuffle
from numpy import array
import argparse
import sys
from sklearn.lda import LDA
from sklearn.preprocessing import MinMaxScaler


def fitAndPredict(trainI, testI, k, feat_vecs, labels):
    """ do one round of fitting to chi2, making model, and
        doing predictions
    on test"""
    lda = LDA(n_components=k)
    best = lda.fit(feat_vecs[trainI].toarray(), labels[
        trainI])
    test_feats = best.transform(feat_vecs[testI].toarray())
          # test data reduced to same k features
    train_feats = best.transform(feat_vecs[trainI].toarray
        ()) # train data reduced to same k features
    train_labels = labels[trainI] # labels for this sample
        section

    #Data can be negative, I need it positive for
        MultinomialNB
    #Scale it:
    mm_scaler=MinMaxScaler()
    scale=mm_scaler.fit(train_feats)
    scaled_test_feats=scale.transform(test_feats)
    scaled_train_feats=scale.transform(train_feats)

    fitted = clf.fit(scaled_train_feats, train_labels)
    pred = fitted.predict(scaled_test_feats) # predict
        labels for test
    return pred

def iterK(max_k,step_size,trainI,testI,i,feat_vecs,labels,
    test_labels):
    """ Search through all k, return prediction of model
        with
    higest F1"""
    f1_k = []
    print("fold---------------> " + str(i))
    for k in range(1,max_k,step_size):
        pred = fitAndPredict(trainI,testI,k,feat_vecs,
            labels)
        if k != 1:
            f1_k.append((f1_score(test_labels, pred), k))
            print("%d features: f1=%.4f"%(f1_k[-1][1], f1_k
                [-1][0]))
    k = sorted(f1_k, key=itemgetter(0))[-1][1]
    pred = fitAndPredict(trainI,testI,k,feat_vecs,labels)
    # compute scores
    k_list.append(k)
    return pred


def xval(svm_light_in,num_folds,num_steps=25,get_best=True,
    k=None):

    global k_list

    feat_vecs,labels = svmlight.load_svmlight_file(
        svm_light_in)

    max_k=feat_vecs.shape[1]
    print('MAX_K: ', max_k)
    step_size = max_k//num_steps
    kf = StratifiedKFold(labels, n_folds=num_folds, shuffle
        =False) # make folds

    f1 = []
    accuracy = []
    precision = []
    recall = []
    k_list = []
    i = 1
    for trainI,testI in kf:
        test_labels = labels[testI]
        if get_best:
            pred = iterK(max_k,step_size,trainI,testI,i,
                feat_vecs,labels,test_labels)
        else:
            pred = fitAndPredict(trainI,testI,k,feat_vecs,
                labels)


        f1.append(f1_score(test_labels,pred))
        accuracy.append(accuracy_score(test_labels,pred))
        precision.append(precision_score(test_labels,pred))
        recall.append(recall_score(test_labels,pred))
        i+=1
    f1 = array(f1)
    accuracy = array(accuracy)
    precision = array(precision)
    recall = array(recall)
    k_list = array(k_list)
    print('Avg␣F1:␣' + str(f1.mean()))
    print('Avg␣Accuracy:␣' + str(accuracy.mean()))
    print('Avg␣Precision:␣' + str(precision.mean()))
    print('Avg␣Recall:␣' + str(recall.mean() ))
    if get_best: print('Avg␣k:␣' + str(k_list.mean()))
    print(f1)
    print(accuracy)
    print(precision)
    print(recall)
    print(k_list)
    return [score.mean() for score in (f1,accuracy,
        precision,recall)]


parser = argparse.ArgumentParser()
#parser.add_argument('posts_id', help='identificador del
    conjunto de posts')
#parser.add_argument('nu', help='factor nu para NuSVC (
    intervalo (0,1])', type=float, default=0.42)
parser.add_argument('-a', '--archivo', help='guardar␣salida
    ␣en␣archivo', action='store_true')
parser.add_argument('--folds', help='n mero␣de␣folds␣[10]'
    , type=int, default=10)
args = parser.parse_args()

if args.archivo:
     sys.stdout = open('stat-MNB-SVM-withFS_LDA_multinomial.
         txt', 'w', buffering=1)

print("Loading␣dataset␣...")
svm_light_in = 'results_stat.txt'

clf = MultinomialNB()

xval(svm_light_in, args.folds, get_best=True, k='all')
```