Applied Statistics

Sitbon Pascal

Dehar Madjer

Hamdaoui Amine

Lecocq Thomas

Machine learning algorithms applied to insult detection in online forums

ENSAE Paris Tech 2013-2014

Contents

INTRODUCTION

Machine learning is a branch of artificial intelligence about constructing and studying systems that can learn from data. In 1959, Arthur Samuel, a famous pioneer in the field of artificial intelligence and computer science, gave a formal and interesting definition of machine learning: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E". This sentence puts the stress on the main principle of machine learning techniques: a general inductive process automatically builds a classifier by learning, from a set of preclassified documents (called training set), the characteristics of the categories. More precisely, our goal is to train a machine learning system on forum messages to learn to distinguish between insulting messages and "clean" messages. In order to study a wider range of classifiers, we decided to illustrate three different classification methods that we discuss later. Rather than being exhaustive and presenting all kind of classifiers, we preferred to take the time to explain all the mathematical background behind the learning methods we chose. Indeed, one has to understand what is going on behind an algorithm if he wants to make it better.

The outline of this document is as follows. **Part 1** starts with some descriptive statistics and provides some background on optimization, regularization which is a crucial concept in machine learning and gradient methods which are used in several of the training methods we describe. **Part 2** gives the basic theory about three different kinds of classification methods: support vector machines, logistic regression and decision tree. **Part 3** tries to go further and to optimize the algorithms obtained in part 2.

I. OPTIMIZATION IN LEARNING

A) Descriptive statistics

We have three different datasets for our work: a small one, a medium one and a large one. We call m the number of observations and q the dimension of each observation –inputs–, then, our dataset is a matrix –called X–with m rows and q columns. Depending on the dataset we choose, q is either 2245, or 16294, or 57509. This diversity is interesting for comparisons and also when it comes to overfitting issues. Each column is proportional to the number of occurrences of a word in the sentence. For instance, the first column could represent the number of occurrences of the word "*fuck*" in the messages. Actually, it is not exactly the number of occurrences since our data is normalized, nonetheless, it is proportional. This modes is called Bag-of-words. For each of the three matrix, $m = 6594$, which means that the number of messages does not change from a database to another; the changes only affect the number of columns. The training dataset is completed with a vector $Y \in \mathbb{R}^{6594}$ classifying every row –i.e. messages– of the matrix X depending on whether it is considered as insulting or not. To illustrate it more formally, the i^{th} value of Y is either 1 if the message corresponding to the i^{th} row of X is considered as insulting or -1 if it is not. Therefore, we can give an analytic expression to our training dataset:

$$T = \{(\overrightarrow{x_l}, y_l), 1 \leq l \leq m, \forall l\ y_l \in \{-1,1\}, \overrightarrow{x_l} \in \mathbb{R}^q\}$$
$\overrightarrow{x_i}$ corresponds to the i^{th} row of X, it can be seen as a message
y_i corresponds to the i^{th} value of Y, it is the label of $\overrightarrow{x_i}$

When we look closely at the repartition of our training dataset, we can see that it is imbalanced. Indeed, there are 1742 insulting messages (26% of total messages) and then 4852 clean messages. Note that such disequilibrium can affect the efficiency of an algorithm, we discuss it in more detail in §II.A)2)v). Note also that

the three matrices at our disposal are **sparse matrices** (a spare matrix stores only the non-zero elements of a dense matrix) and it has several advantages:

- It takes less space than dense matrices especially when there are many "0".
- It enables algorithms to be much faster since they can scan matrices and make operations quickly.

B) Generalization error

Supervised learning is the machine learning task of inferring a function from labeled training data. To describe our problem formally, our goal is to learn a function $f: X \rightarrow Y$ given a training set, so that $f(x)$ is a good predictor for the value of y associated to x (Figure 1.1). Furthermore, a key objective for a learner is **generalization**. Generalization, in this context, is the ability of a learning algorithm to perform accurately on new and unseen data after having experienced a learning data set. In our case, we would like the learner to return 1 or -1 for any new vector x –corresponding to a new message– depending on the nature of the message. We can try to illustrate the process:

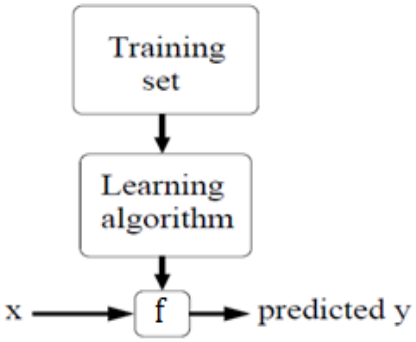

Figure 1.1: The learning process.

The simplest way to predict y is to consider the function f as a linear function of $\vec{x} \in \mathbb{R}^q$:

$$f_{\vec{w},b}(\vec{x}) = <\vec{w},\vec{x}> + b = \sum_{i=0}^{q} w_i x_i + b, \qquad b \in \mathbb{R}, \vec{w} \in \mathbb{R}^q$$

Where $<,>$ designs the usual Euclidian inner product on $\mathbb{R}^q$

Here, the vector $\vec{w}$ is the parameter, or weight, parameterizing the space of linear functions mapping from X to Y. The aim of our work is to approach the target function that would make no mistake in its predictions. f can obviously take more values than +1 and -1, but we will say that when $f_{\vec{w},b}(\vec{x}) \geq 0, \vec{x}$ belongs to the same class than the ones for which $y_l = 1$, and the other way around for the other class. Now that we have a training set and a hypothesis (the function *f*), how do we learn the parameter $\vec{w}$? We must have a measure of how efficient a learning algorithm is in regards to a specific learning problem. One of the most common criterions for machine learning algorithms is **the generalization error** which enabled us to estimate the efficiency of algorithms on future data. Usually, we want to lower the number of misclassifications as much as possible. A misclassification is when our function f fails to predict the correct value of y associated to x, i.e. *f(x)≠y* for a given pair (x, y) of the training data.
We can define the generalization error of a classifier:

$$g(\vec{w}) = E[1_{f(x)\neq y} | (x,y)\sim D]$$

where D is the joint distribution over $X \times Y$ of the data points $x \in X$ with the true labels $y \in Y$.

Finding an analytic expression of g is key since it makes it easy to find the vector $\vec{w}$ that defines our function f. Indeed it will be the vector that minimizes the generalization error. However, we generally do not know the distribution D of points. So we try to minimize a function $\hat{g}$ on the training set $T = \{(\vec{x_l}, y_l), 1 \leq l \leq m, \forall l\ y_l \in \{-1,1\}, \vec{x_l} \in \mathbb{R}^q\}$. The first idea we had was to minimize the $0-1$ loss on the training dataset but this problem is quite difficult since the loss function is not convex. In order to use well-known algorithms for optimization, we need $\hat{g}$ to be a convex function.

Then we chose a usual analytic expression for $\hat{g}$:

$$\hat{g}(\vec{w}) = \frac{1}{m}\sum_{i=1}^{m} l(\vec{x_i}, y_i, \vec{w})$$

Where l is a convex loss function that approximates the $0-1$ *loss.*

This function is also called the **empirical loss**. It should be underlined that $\hat{g}$ is only an estimator of the true generalization error and minimizing $\hat{g}$ does not necessarily imply minimizing the real generalization error. Sometimes, when learning is performed too long, the algorithm may adjust to very specific random features of the training data that have no causal relation to the target function. Then the training error decreases and the validation error increases while the percentage of the dataset used to constitute the training dataset increases (i.e. the learning algorithm trains more). This phenomenon is called **overfitting** (Figure 1.2).

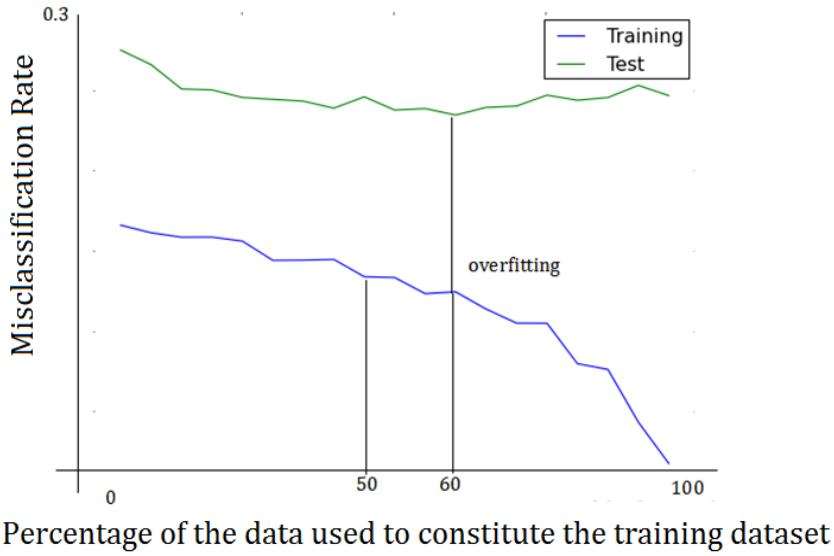

Figure 1.2: Misclassification on test and training set depending on the % of data used to constitute the training data.

The concept of overfitting is important in machine learning. Overfitting happens when the vector $\vec{w}$ is too complex (i.e. when its norm is too larger). Then the vector $\vec{w}$ is too specific to a model and the learning algorithm will give us low results on unseen data. One usual way of avoiding overfitting is **regularization**.

C) Regularization and Gradient method

Regularization consists in adding an extra term to the error function $\hat{g}$ in order to penalize complex $\vec{w}$ vectors. Let's call r this function, note that it is a function of $\vec{w}$. The function r can take different forms, most of the time it is the l_1 or l_2 norm of $\vec{w}$. Even if it can seem arbitrary, it can be shown that it minimizes the influence of spurious correlations in the training set. Indeed, it limits the complexity of the model since $\|\vec{w}\|$ would be larger when vector $\vec{w}$ is more complex. Our new error function can be defined as follows:

$$\hat{g}(\vec{w}) = \frac{1}{m}\sum_{i=1}^{m} l(\vec{x_i}, y_i, \vec{w}) + r(\vec{w})$$

Now that we have our optimization problem, we need a method to minimize $\hat{g}$ i.e. to find $\vec{w} = argmin\ \hat{g}(\vec{w})$. A classical mathematical method to find the minimum of a function is gradient descent. This method is based on the fact that the gradient ∇f of a function f points in the direction of the greatest increase so that $-\nabla f$ points in the direction of the greatest decrease. Then a natural iterative algorithm is to update an estimate $\vec{w}$:

$$w_i := w_i - \alpha \frac{\partial \frac{1}{m}\sum_{l=1}^{m} l(\vec{x_l}, y_l, \vec{w})}{\partial w_i}$$

This update is simultaneously performed for all values of $i = 0, \ldots, q$

Here, α is the learning rate, it determines how far we move in the direction of the gradient. It is a key parameter. Indeed, it should not be too small if we do not want the algorithm to be too long and it should not be too large if we do not want to miss the minimum. This method looks at every example in the entire training set on every step, and is called the **Batch gradient descent**. The ellipses shown below are the contours of a quadratic function (Figure 1.2). Also shown is the trajectory taken by the gradient descent method.

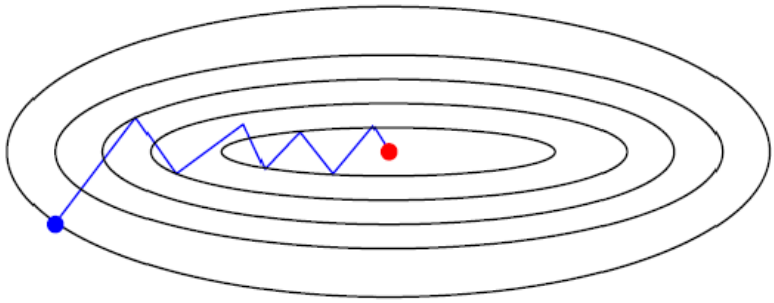

Figure 1.3: The gradient descent trajectory. (Adapted from reference [1])

Note that the gradient descent is deterministic, which means that every time we run for a given training set, we will get the same optimum in the same number of iterations. However, when our training sample is very large, this method may take too long because in every iteration we are running through the complete training set.

There is an alternative to the Batch gradient descent that also works very well and avoid running through the entire training set, consider the following algorithm:

$$w_i := w_i - \alpha \frac{\partial l(\vec{x_l}, y_l, \vec{w})}{\partial w_i}$$

Where l *is an index drawn randomly at each step from* $\{1, 2, \ldots, m\}$

This update is simultaneously performed for all values of $i = 0, \ldots, q$

In the gradient descent, we compute the gradient using the entire training set. In this algorithm, we approximate the gradient by using a single example of the training data set. This technique is called the stochastic gradient descent method. We run through the training set and every time we find a training example, we update the parameters according to the gradient of the error with respect to that single training example only. As we use only a single example each time we are only getting an approximation to the true gradient, then we do not have any guarantee of moving in the direction of the greatest descent. Nevertheless, there are at least two reasons why stochastic gradient descent is always useful for large-scale learning problems. First, it is way quicker than the Batch gradient descent method when m is large. On the other hand, it can be shown that the stochastic gradient descent often gets close to the minimum much faster than the usual gradient descent.

Actually, it may never "converge" to the minimum since it uses an approximation of the gradient so the parameter $\vec{w}$ will keep oscillating around the minimum; yet, in practice, the values near the minimum will be good-enough approximations of the real minimum. For instance, the trajectory shown below is the trajectory taken by the stochastic gradient descent method (Figure 1.3).

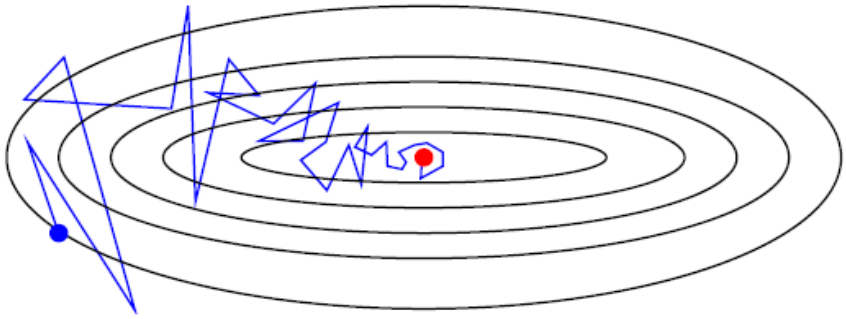

Figure 1.4: The stochastic gradient descent trajectory. (Adapted from reference [1])

In 2008, Leon Bottou (Princeton) and Olivier Bousquet (Google) published a paper called *The Tradeoffs of Large Scale Learning* to show concretely how a learning algorithm can be poor at optimization but excellent when it comes to generalization. They developed a framework that explicitly includes the effect of optimization error in generalization and then they analyzed the asymptotic behavior of this error when the number of training examples is large. They did it by looking at the estimation error which is intuitively a measure of how representative of the implicit distribution the training set is. When they compared the results obtained by the Batch gradient descent and the results obtained thanks to the stochastic gradient descent, their conclusion was that the stochastic gradient algorithms benefit from their lower estimation error in large datasets to achieve a low generalization error quicker than basic gradient descent algorithm. One of the key interrogations of this paper is the question of how useful approximate solutions are to learning problems: what is the cost in term of generalization errors if we only approximately solve the optimization problem that models our goal? The authors answered this question by showing that we do not need to focus on precise solutions –in large-scale learning–since it is possible to achieve low generalization error without deterministic solutions. This is something interesting –one can have a look at the original document for more mathematical details- and should be underlined since it is not common in computer science. Indeed, most of the time, the optimization problem is the problem of interest while in learning the real quantity of interest is the generalization error. As we saw, the optimization formulation of the problem is only a compensation for the fact we never know the underlying distribution of the test data. Actually, this is one of the key contributions of Bottou and Bousquet's paper.

Even though stochastic gradient descent has been around in the machine learning community for a long time, it has received a considerable amount of attention in the context of large-scale learning just recently. Indeed, its iteration complexity is independent of n while the Batch gradient descent complexity is linear in n. Most of the algorithms in the methods we study such as SVM and logistic regression use variants of the stochastic gradient descent. This is something clear when we look at Scikit-Learn algorithms. Scikit-Learn is an open source machine learning library for the Python programming language that we use for our work.

However, the stochastic gradient descent has a few drawbacks:

- It is sensible to feature scaling which is not an issue for us since our dataset is already normalized.
- The l loss function and the regularization function must be convex functions.
- It requires a number of parameters such as the regularization function and the number of iterations.

II. SOME WELL-KNOWN CLASSIFIERS

NB: In II) and III) we will use the same training set and the same test set in order to compare the different classifiers. In order to do so, we set a random seed in the algorithm used for creating the training and test datasets. A random seed is a number used to initialize a pseudorandom number generator. We used the seed "0" throughout this paper. We decided use 60% of the data set for the training set and 40% for the test set as it is usually done for datasets that contain approximately as many data points as our data set.

This part will present three of most well-known classifiers: Support vector machines, decision trees and logistic regression. For each classifier we will some key points of the theory necessary to understand the simulations and how the classifier is obtained.

The two most common indicators that allow oneself to evaluate the efficiency of a classifier on a set are the misclassification rate and the recall rate. The first one is the number of messages that wrongly predicted in this set:

Misclassification Rate: $\frac{\text{Number of insults classified as clean+Number of clean messages classified as insults}}{\text{Number of insulting messages in the set}}$

Accuracy = $1 -$ Misclassification Rate

Here we are tracking insults; the recall rate therefore indicates the percentage of insults captured in this set:

Recall Rate: $\frac{\text{Number of insulting messages}-\text{Number of insults classified as clean messages}}{\text{Number of insulting messages in the set}} = \frac{\text{Number of insults well predicted}}{\text{Number of insults}}$

A) Support Vector Machines

The principal goal of SVM is to find a "good" separation between the two classes (insulting or clean messages for us); this separation is then used to make predictions. Indeed, messages from a training data set are mapped into a specific space (which axes are the variables in our matrix), the algorithm finds a hyperplan which separate the messages of the two classes in a "good" way. Then if a new message is given, his predicted class will depend on his position with regard to the hyperplan.

1) Theory

As written in part I, a data set containing points which belong to two classes can be represented by the following set:

$$D = \{(\vec{x}_l, y_l), 1 \leq l \leq m, \forall l\ y_l \in \{-1,1\}, \vec{x}_l \in \mathbb{R}^q\}, (m,q)\mathbb{N}^2$$

$\vec{x}_l$ is a message for $1 \leq l \leq m$
y_l represents the belonging to one of the two classes ($y_l = 1$ if $\vec{x}_l$ is an insult, else $y_l = -1$)
m is the number of data points (Number of messages)
q is the number of features observed for each message

The easier example to give is a linearly separable set, then we will see how to deal with a data set which points are not linearly separable, and even not separable (which is more often the case when the data set contains a reasonable number of points)

- **Linearly separable set**

On Figure 2.1 it's easy to see that the data points can be linearly separated. But, most of the time with a big data set, it's impossible to say just by visualizing the data that it can be or not linearly separated, even the data can't be visualized in fact.

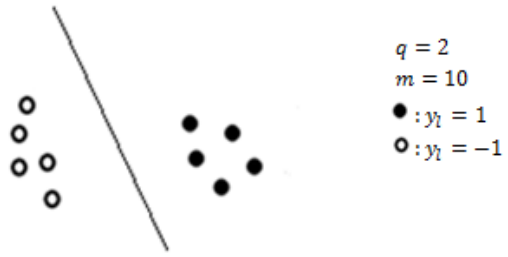

Figure 2.1: A linearly separable set (D).

Let's give some definitions necessary to understand and solve the problem analytically:

Notation: $<,>$ will design the usual Euclidian inner product on $\mathbb{R}^q$ and $||.||$ the associated norm.

Definition: A linear separator of $\mathbb{R}^q$ is a function f which depends on two parameters that we will note $\vec{w} \in \mathbb{R}^q$ and $b \in \mathbb{R}$. f is given by the following formula:

$$\forall \vec{x} \in \mathbb{R}^q, f_{\vec{w},b}(\vec{x}) = < \vec{w}, \vec{x} > + b, b \in \mathbb{R}, \vec{w} \in \mathbb{R}^q$$

This separator, can take more values than +1 and -1, but we will say that, when $f_{\vec{w},b}(\vec{x}) \geq 0, \vec{x}$ belongs to the same class than the ones for which $y_l = 1$, same comment for the other class. The line of separation is the contour line defined by: $f_{\vec{w},b}(\vec{x}) = 0$.

Definition: The margin of an element $(\vec{x}_l, y_l) \in \mathbb{R}^q \times \{-1,1\}$ relatively to a separator$f$, noted $\gamma^f_{(\vec{x}_l,y_l)}$, is a real given by the following formula:

$$\gamma^f_{(\vec{x}_l,y_l)} = f(\vec{x}_l) * y_l$$

Definition: The margin is linked to a separator f and a set of points D, it is the minimum of the margin of each couples $(\vec{x}_l, y_l) \in D$:

$$\gamma^f_D = \min\{\gamma^f_{(\vec{x}_l,y_l)}, (\vec{x}_l, y_l) \in D\}$$

Definition: The support vectors are the ones for which: $y_l(< \vec{x}_l, y_l > + b) \leq 1$, i.e. : $\gamma^f_{(\vec{x}_l,y_l)} \leq 1$.
Intuitively, the support vectors are the ones which have an impact on the hyperplan boundary (Figure 2.2), removing or moving support vector will change the equation of the sparating hyperplan.

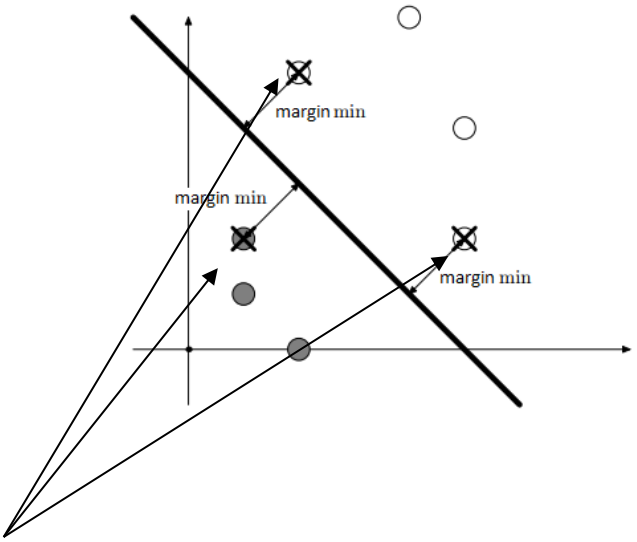

Figure 2.2: The support vectors are marked with a cross; removing one of them will change hyperplan boundary- Adapted from reference [15].

The goal of the SVM is to find a hyperplan which maximizes the margin, this leads to the following optimization problem:

$$\text{Min}_{\vec{w},b}\frac{||\vec{w}||^{2}}{2}$$
$$\text{u. c.}\left(\gamma^{f}_{(\vec{x}_l,y_l)} \geq 1, \forall(\vec{x}_l,y_l) \in D\right) \ (*)$$

NB: We minimize $\frac{||\vec{w}||^{2}}{2}$, instead of $||\vec{w}||$, because it simplifies the calculus (differentiations, and it is always better to work with the square norm).

- **Inseparable set**

For this set (Figure 2.3) of data points, any linear classification would introduce too much misclassification to be considered as accurate enough.

Figure 2.3: An example of set which can't be linearly separable.

Projection into a bigger space:

A solution to get a better separation is to project each points of D in a bigger space (in terms of dimension), and to make a linear separation into the new space. Let's name φ this projection:

$$\forall(\vec{x}_l, y_l) \in D, \varphi(\vec{x}_l) = \begin{pmatrix} \varphi_1(\vec{x}_l) \\ . \\ . \\ . \\ \varphi_n(\vec{x}_l) \end{pmatrix}$$

This point of view can lead to problems, because n can grow without limit, and nothing assures that the φ_i are linear. Following the same method than above would imply to work on a new set:

$$D^* = \varphi(D) = \{(\varphi(\vec{x}_l), y_l), 1 \leq l \leq m, \forall l, y_l \in \{-1,1\}\}$$

This method will be never used because it implies to calculate φ for each vectors of D. Let's notice that it's not necessary to calculate φ, indeed the optimization problem only involved inner products between the different vectors. Let's note $k(\vec{z}, \vec{x})$ the inner product: $< \varphi(\vec{z}), \varphi(\vec{x}) >$, the trick is that we will first give the function k making sure that it corresponds to a projection in an unknown space (such a function is called a kernel), it enables us to avoid the calculus of $\varphi(\vec{x})$, we won't also try to describe the space where we are projecting. The optimization problem is the same than for the linearly separable set (page 10), we just have to replace $<\cdot,\cdot>$ by $k(\cdot,\cdot)$:

$$\text{Min}_{\vec{w},b} \frac{k(\vec{w}, \vec{w})}{2}$$

$$y_l(k(\vec{x}_l, \vec{w}) + b) \geq 1, \forall(\vec{x}_l, y_l) \in D (**)$$

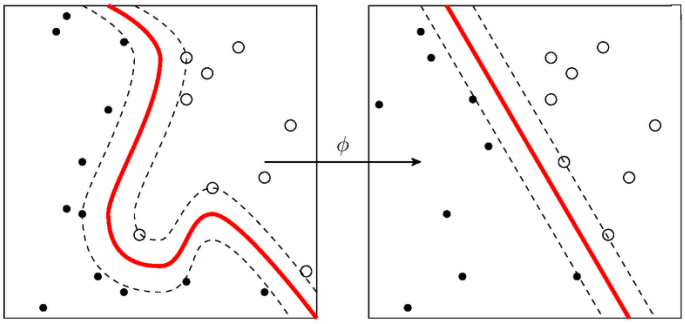

Figure 2.4: Kernel Trick – Projection into another space.

It's not sure that a perfect separation between the two classes exists in any spaces. Consequently, we allow some data points to have a margin smaller than 1 (even negative when some points are not well classified), $(**)$ become:

$$y_l(k(\vec{x}_l, \vec{w}) + b) \geq 1 - \varepsilon_l, \forall(\vec{x}_l, y_l) \in D, \varepsilon_l \in \mathbb{R}_+^*$$

We also changed the function that we are minimizing, because we can minimize $\frac{k(\vec{w},\vec{w})}{2}$ until it's equal to 0 by choosing the ε_l high enough. So we add an error term: $C\sum_{1\leq l\leq m}\varepsilon_l$ in order to correct this problem. Eventually, the optimization problem is :

$$\text{Min}_{\vec{w},b,\varepsilon_l}\frac{k(\vec{w},\vec{w})}{2}+C\sum_{1\leq l\leq m}\varepsilon_l$$
$$y_l(k(\vec{x}_l,\vec{w})+b)\geq 1-\varepsilon_l, \forall(\vec{x}_l,y_l)\in D, \varepsilon_l\in\mathbb{R}_+^*$$

2) Simulations

- **Linear Kernel function**

As expected there is no reason that a linear separator exists in the initial space. For many values of C, we observed accuracy around 75%, but the first kind error (messages classified as clean, but which are insults) constituted the whole part of the error. That's not really good and can really be improved using other Kernel functions which correspond to bigger spaces. We decided to use the Radial Basis Kernel function.

- **Radial Basis Kernel function**

(i) The radial basis kernel function

$$\forall(\vec{z},\vec{x})\in\mathbb{R}^q\times\mathbb{R}^q,\quad k(\vec{z},\vec{x})=e^{-\frac{||\vec{x}-\vec{z}||^2}{2*\gamma}}$$

It is well known that this function corresponds an inner product between the projected of $\vec{z}$ and $\vec{x}$ in an infinite dimensional space. Intuitively, it will be easier to find a separation in a bigger space (in terms of dimension), that's the principal reason that led us to choose this kernel function.

(ii) How to optimize C and γ:

The module Scikit-Learn enables us to draw graphics which show the accuracy of the separator on the training set for different values of γ and C (a grid). The first step was to find an area in this grid, where the accuracy was high enough. The following graphic (Figure 2.6) represents the misclassification rate made on the training set for different values of γ and C. The scale on the right of the graphic indicates to which accuracy corresponds each color in the graphic.

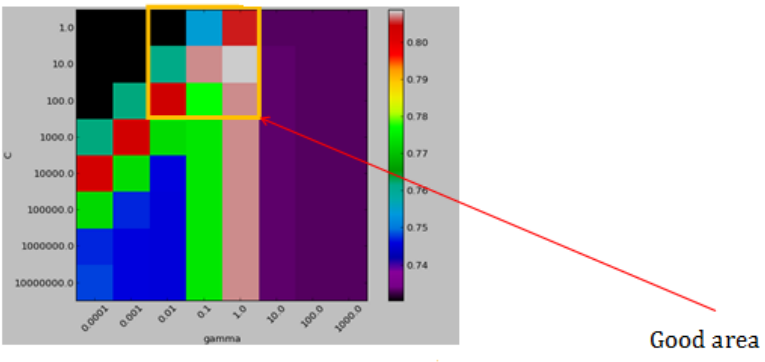

Figure 2.5: Accuracy on training set depending on different values of the color represent the γ and C.

Since we found good parameters in the area found on the first grid (Figure 2.5), we now look for the best parameter of this area (see pseudo-code below) in order to find better parameters. Indeed there is no reason that the parameters found are the better and looking around the first "good" parameters should help us to improve the accuracy of the separator.

(iii) Maximum Research algorithm

Input: N (number of iterations), Training data set: $\{(\vec{x}_l, y_l), 1 \leq l \leq m, \forall l\ y_l \in \{-1,1\}, \vec{x}_l \in \mathbb{R}^q\}$ $(C_{initial}, \gamma_{initial})$ *(The best parameters of the good area found before).*

Initialize $(C_{res}, \gamma_{res}) = (C_{initial}, \gamma_{initial})$

For j from 1 to N:

$C_{range} = [C_{res} * 10^{-\frac{1}{2^j}}, C_{res}, C_{res} * 10^{\frac{1}{2^j}}]$

$\gamma_{range} = [\gamma_{res} * 10^{-\frac{1}{2^j}}, \gamma_{res}, \gamma_{res} * 10^{\frac{1}{2^j}}]$

Find best parameters in $C_{range} \times \gamma_{range}$: $(C_{best}, \gamma_{best})$ *using cross validation (Stratified-kfold=2)*

$(C_{res}, \gamma_{res}) = (C_{best}, \gamma_{best})$

Return (C_{res}, γ_{res})

Stratified Kfold=2: When looking for the best parameters, the training set is divided into two data sets, each set containing the same percentage of samples of each target class as the complete set. Then the parameters in $C_{range} \times \gamma_{range}$ are tested on these sets, considering them as Training and Test set, and vice versa.

Figure 2.7 shows how the accuracy changed depending on the number of iterations of the algorithm. It increased from 83.9% to nearly 84.5% for the test set at the end and the accuracy on the training also increased by 0.5%.

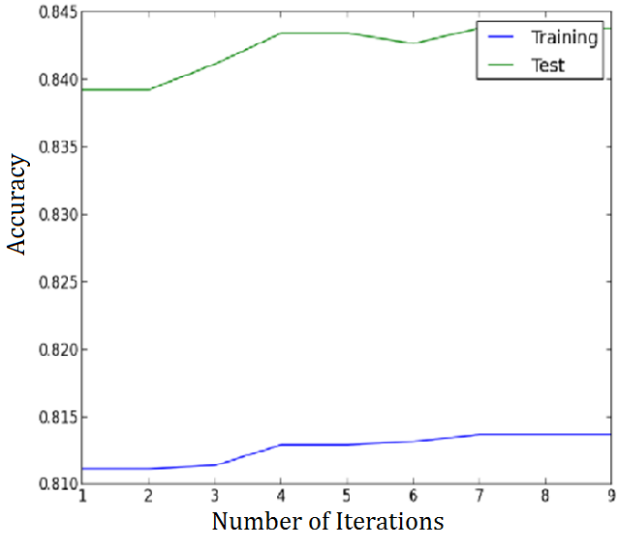

Figure 2.6: Accuracy of the separator on the test and on the training set at each step of the algorithm

These results are good but can be improved in many ways. We are tracking down insults so we are more interested in reducing the number of insults classified as non-insulting messages (**First kind error**). Note the first kind error was equal to 294 which is not really good because there are 675 insults in the data set, it means that only 51% of the insults were detected. Keeping a good misclassification rate, it would be good to reduce the first kind error. There is a way to attribute a larger cost to insults misclassifying, as explained below. This method will help us to reduce the first kind error while keeping a good overall accuracy.

(iv) Putting weights on error terms

Theory:

Many Data sets (including the ones we are working on) are imbalanced. It means that one class contains a lot more examples than the other one. The principal problem linked to these data sets is that we can no longer say that a classifier is efficient just by looking at the total misclassification rate.

Indeed, let's say that the ratio is 99% (class: +1) against 1%(class: -1). A classifier which misclassifies every vector which belong to class +1, but well classify the vectors of the other class will return a 99% accuracy. Nevertheless if you are especially interested in the other class in your study, this separator won't be very useful. There are several ways to avoid this problem, we will treat one of the most well-known: Different costs for misclassification to each class. What follows is a presentation of how it works analytically and how it can be implemented in Python thanks to the module Scikit-Learn. We will also study Ensemble method in III) but these methods are different because they aggregate many classifier, here the goal is to make the algorithm be more careful with Insults misclassifying. Let's consider a Data set which is unbalanced:

$$D = \{(\vec{x}_l, y_l), 1 \leq l \leq m, \forall l, y_l \in \{-1,1\}, \vec{x}_l \in \mathbb{R}^q\}, (m,q)\mathbb{N}^2$$

Here is the optimization problem solved by SVM:

$$\text{Min}_{\vec{w},b,\varepsilon_l} \frac{k(\vec{w},\vec{w})}{2} + C \sum_{1 \leq l \leq m} \varepsilon_l \, , C \in \mathbb{R}_+^*$$

$$y_l(< \vec{x}_l, y_l > +b) \geq 1 - \varepsilon_l, \forall (\vec{x}_l, y_l) \in D (**)$$

The trick is to replace the total misclassification cost: $C \sum_{1 \leq l \leq m} \varepsilon_l$ by a new one:

$$C_+ \sum_{j \in J_+} \varepsilon_j + C_- \sum_{j \in J_-} \varepsilon_j \, , C_+ \geq 0, C_- \geq 0$$

$$J_+ = \{j \in [|1,m|], y_j = +1\}$$

$$J_- = \{j \in [|1,m|], y_j = -1\}$$

One condition has to be satisfied, in order to give equal overall weight to each class; the total penalty has to be the same for each class. A hypothesis commonly made is to suppose that the number of misclassified vectors in each class is proportional to the number of vector in each class, which leads us to the following condition:

$$C_+ * card(J_+) = C_- * card(J_-)$$

It shows that, if for instance: $card(J_+) \gg card(J_-) \Rightarrow C_+ \ll C_-$. Indeed a larger importance will be given to misclassified vectors $\vec{x}_l$ for which $y_j = -1$.

Implementation explanations:

The option for weighting is class.weight which hast to be a dictionary in Python. But what can be disappointing is that you also have to enter a cost. What does mean this cost?
Let's name ω_1 and ω_{-1} the respective weights for each class, C the Cost. What becomes C_+ and C_- ?
The formula underneath is:

$$C_+ = \omega_1 * C$$

$$C_- = \omega_{-1} * C$$

And Python solves the optimization problem:

$$\text{Min}_{\vec{w},b,\varepsilon_l} \frac{k(\vec{w},\vec{w})}{2} + C_+ \sum_{j \in J_+} \varepsilon_j + C_- \sum_{j \in J_-} \varepsilon_j \, , C_+ \geq 0, C_- \geq 0$$
$$y_l(< \vec{x}_l, y_l > + b) \geq 1 - \varepsilon_l, \forall (\vec{x}_l, y_l) \in D(**)$$

Simulations:

The results were better (Table 2.1); we always had a much lower first kind error compared to (iii), keeping a good overall accuracy. For instance, for the same training and test set (the test set contains 2638 messages including 675 insults). For the first algorithm (without weights) the first kind error was 289 and the second one was 124 on the test set. Putting weights enabled us to reduce the first kind error from 289 to 194 (which represent around 14% of the total of insulting messages in the test set) and the second one increased from 124 to 293. Our goal is to track down insults on forums, so putting weights is a good idea because we well predicted nearly all the messages that were insulting. Note that the recall rate ($\frac{\text{number of insulting messages in test set} - \text{1st kind error}}{\text{number of insulting messages}}$) improved by 14% using weights.

	Accuracy	Recall Rate
SVM without weights	84%	57%
SVM with weights	82%	71%

Legend: *Recall Rate* $\geq 70\%$, *Recall Rate* $\geq 50\%$ *Recall Rate* $< 50\%$

Table 2.1: Misclassification Rate and Recall Rate on Test Set.

B) Decision Trees:

1) Theory

A decision tree is a decision support tool that uses a tree-like graph and their possible consequences. Decision tree learning uses a decision tree as a predictive model which maps observations about an item to conclusions about the item's target value. It is another of the predictive modeling approaches used in machine learning.

Classification trees partition $\mathbb{R}^d$ into regions, often hyper-rectangles parallel to the axes (Figure 2.8). Among these, the most important are the binary classification trees, since they have only two different classes to predict and are thus easier to manipulate and update. The top of a binary tree is called the root. Each node has either no child and in that case is called a leaf or a terminal node, or two children. The height of a tree is the maximal depth of any node. Each node represents a set in the space $\mathbb{R}^d$. If a node w represents the set A and its children w, w' represent A' and A'', then we require that $A = A' \cup A''$ and $A' \cap A'' = \emptyset$. The root represents $\mathbb{R}^d$ and the leaves, taken together, form a partition of $\mathbb{R}^d$.We associate each class in some manner with each leaf in a classification tree. If a leaf represents a region A and we call our classification tree "f", then for every $\vec{x} \in A$:

$$f(\vec{x}) = \begin{cases} 1 \text{ if } \sum_{i\,:\,\vec{x}_i \in A} y_i > 0 \\ -1 \text{ otherwise} \end{cases}$$

That is, in every leaf region, we take a majority vote over all $(\vec{x}_l, y_l)$ with $\vec{x}_l$ in the same region. Ties are broken in favor of class "-1". One of the most compelling reasons for using binary tree classifiers is to explain complicated data and to have a classifier that is easy to analyze and understand (Figure 2.9).

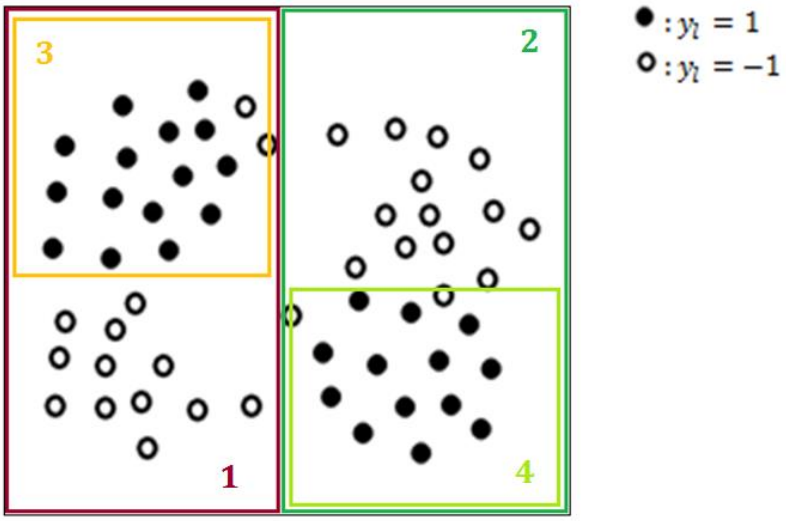

Figure 2.8: Example of decision surfaces of a decision tree.

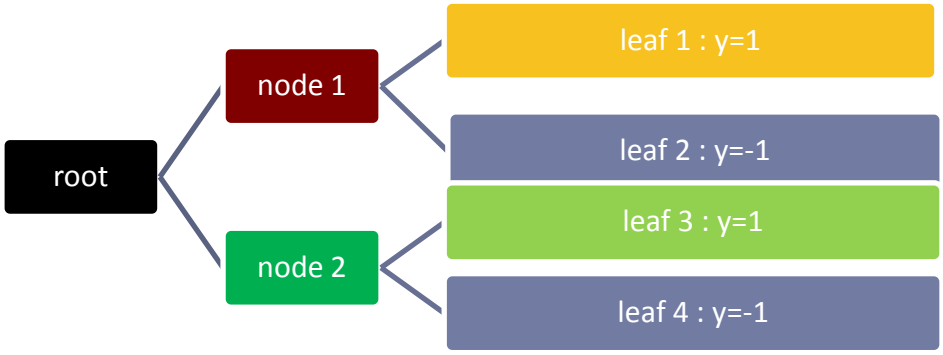

Figure 2.9: Example of a decision tree.

For computational reasons, classification trees are produced by determining the splits recursively. At any given stage of the tree-growing algorithm, some criterion is used to determine which node of the tree should be split next and where the split should be made, and the procedure is applied recursively to the subtrees. Nowadays, classification trees are mainly created using the CART program, as we did for this paper. One of its key ideas is the notion that trees should be constructed from the bottom up, by combining small subtrees. The starting point is a tree with *m+1* leaf regions defined by a partition of the space based on the *m* data points. When constructing a starting tree, a specific splitting criterion is applied recursively. The criterion depends only on the coordinatewise ranks of the point and their labels, and will determine which rectangle should be split and where the cut should be made. The algorithm uses a function for a possible split, so that the splits create the most homogeneous subtrees possible regarding that function. The Shannon entropy and the Gini function – which we will only focus on– are the most commonly used functions for determining the homogeneity of a cluster.

2) Simulation

The accuracy of the simple binary trees remains around 73%, while the recall rate stays around 50%. However, as there is some randomness in the creation of the trees. The accuracy and recall rates of the classification trees varie even when a seed is set for the creation of the data and training sets. For this reason, the algorithm was run five times and the results were computed below (Table 2.2):

Iteration	Accuracy	Recall rate
1	74%	51%
2	73%	51%
3	73%	49%
4	74%	50%
5	73%	51%
Mean (variance)	$73\%(7.4\times 10^{-4})$	$50\%(7.2\times 10^{-3})$

Legend: *Recall Rate* $\geq 70\%$, *Recall Rate* $\geq 50\%$ *Recall Rate* $< 50\%$

Table 2.2: Accuracy and recall rate of the classification trees on the test set.

C) Logistic regression

1) Theory

(i) Presentation of the model

To summarize quickly, we have a data set containing points belonging to two classes. It can be represented by the following set:

$$D = \{(\vec{x}_l, y_l), 1 \leq l \leq m, \forall l\ y_l \in \{-1,1\}, \vec{x}_l \in \mathbb{R}^q\}, (m,q)\mathbb{N}^2$$

$\vec{x}_l$ is a message for $1 \leq l \leq m$
y_l represents the belonging to one of the two classes ($y_l = 1$ if $\vec{x}_l$ is an insult, else $y_l = -1$)
m is the number of data points (Number of messages)
q is the Number of features observed for each message

We wish to predict Y the outcome variable *(Y=-1 or 1)* given $= {}^t(X_1, \dots, X_q)$, the explanatory variables. The Logistic Regression classifier is based on the comparison between $P(Y = 1|X)$ and $P(Y = -1|X)$. It assigns to Y the value for which this probability is the highest.

Assuming that $P(X) = P(Y = 1|X)$ and $1 - P(X) = P(Y = -1|X)$:
Since $Y \in \{-1,1\}$, $E(Y|X) = P(X) \in [0,1]$, we cannot directly use a linear model to estimate: $E(Y|X) = P(X) = \beta_0 + {}^tX\beta$ (under the exogeneity assumption: $E(\text{error term}|X) = 0$) because we don't know if if $\beta_0 + {}^tX\beta \in [0,1]$.

To solve this issue, we will assume $P(X) = F({}^tX\beta)$ where F is a known function strictly increasing and which is a bijection from $\mathbb{R}$ to $]0,1[$. We have $F^{-1}(P(X)) = \beta_0 + {}^tX\beta$.

We use here the function $F(x) = \frac{1}{1+exp(-x)}$ which is equivalent to $F^{-1}(u) = ln\frac{u}{1-u}$ (known as the **logit transformation**), so that our model is:

$$ln\frac{P(X)}{1-P(X)} = \beta_0 + {}^tX\beta$$

$\beta_0 \in \mathbb{R}\ and\ \beta \in \mathbb{R}^q$ are the parameters of the linear regression $X \in \mathbb{R}^q$ is the vector of regressors, we thus have:

$$P(X) = \frac{exp(\beta_0 + {}^tX\beta)}{1+exp(\beta_0 + {}^tX\beta)} \text{ and } 1 - P(X) = \frac{1}{1+exp(\beta_0 + {}^tX\beta)}$$

We should then predict $Y = 1$ when $P(X) \geq s$ (typically 0.5) and $Y = -1$ when $P(X) < s$. This means guessing 1 whenever $\beta_0 + {}^tX\beta \geq b \in R$ and -1 otherwise. So logistic regression gives us a linear classifier. The decision boundary separating the two predicted classes is the solution of $\beta_0 + {}^tX\beta = b$, which is a point if X is one dimensional, a line if it is two dimensional, etc…

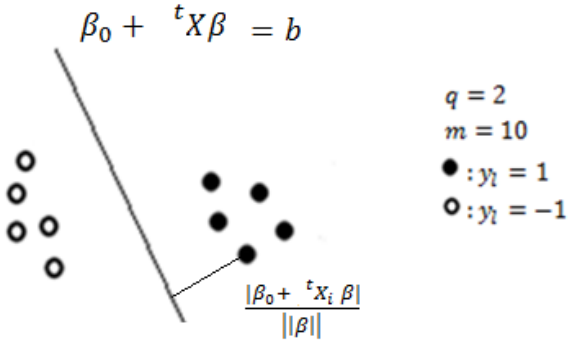

Figure 2.9: A visualization of the classification problem solved by logistic regression.

The distance from a point X_i to the decision boundary is given by $\frac{|\beta_0 + {}^tX_i\beta|}{||\beta||}$ (Figure 2.9). So the farer X_i is from the decision boundary, the higher its class probability is.

(ii) Estimation of the model

Since it is a parametric model, the logistic regression coefficients β_0 and β are typically estimated by maximum-likelihood.

The likelihood function is:

$$L(\beta_0, \beta) = \prod_{i=1}^{m} P(X_i)^{Y_i} (1 - P(X_i))^{1-Y_i}$$

The log-likelihood function is:

$$l(\beta_0, \beta) = \sum_{i=1}^{m} Y_i \, ln\big(P(X_i)\big) + (1 - Y_i) \, ln\big(1 - P(X_i)\big)$$

$$= \sum_{i=1}^{m} Y_i \, ln\left(\frac{exp(\beta_0 + {}^tX_i\beta)}{1 + exp(\beta_0 + {}^tX_i\beta)}\right) + (1 - Y_i) \, ln\left(\frac{1}{1 + exp(\beta_0 + {}^tX_i\beta)}\right)$$

$$= \sum_{i=1}^{m} Y_i(\beta_0 + {}^tX_i\beta) - ln\,(1 + exp(\beta_0 + {}^tX_i\beta))$$

$$l(\beta_0, \beta) = \sum_{i=1}^{m} Y_i(\beta_0 + {}^tX_i\beta) - ln\,(1 + exp(\beta_0 + {}^tX_i\beta))$$

Our optimization program is:

$$Max_\beta \, l(\beta_0, \beta)$$

The logistic regression can also be subject to the overfitting problem described in the previous parts: a large variability in the estimates produces a prediction formula for discrimination with almost no power. A solution to the problem of overfitting exposed in the previous part is penalization. The key idea in penalization is that overfitting is avoided by imposing a penalty on large fluctuations on the estimated parameters and thus on the fitted curve. Another issue with linear models can be the colinearities between the explanatory variables: a remedy is the use of a quadratic regularization ("ridge regularization").

We have the following optimization problem:

$$Min_{\beta} -l(\beta_0, \beta) + \frac{1}{C} * ||\beta||^2$$

Where $C \in \mathbb{R}_+^*$ is a parameter modeling the penalty that discourages high values of the elements of β. The smaller C is, the stronger is the regularization.

This program cannot be solved analytically: numerical methods such as Gradient methods described in the first part are needed to obtain an approached solution.

2) Simulations

(i) Optimizing *C* using cross validation

We are now willing to find the "best" value of *C*, which means the one that minimizes the misclassification rate, here are the algorithms used in order to solve this problem.
We use this "accuracy" function to define our score:

$$f_{0_1}(Y, \hat{Y}) = 1 - \frac{1}{n} * \sum_{i=1}^{n} 1\{Y_i = \hat{Y}_i\}$$

$\hat{Y}$: Prevision of the model, *Y*: True values, *n*: number of messages of the test set.
We began by performing a large search for *C* (see the algorithm here below): we tested all the values between 0.1 and 100 with steps of 0.1 and we obtained the best score for $C_{initial}$=5.3.

Input: X_{train}, Y_{train} *(the training set) and the list* C_{values} *(the values of C we want to test)*

The training set X_{train}, Y_{train} *is divided in 4 subsets* $X_{train}(j), Y_{train}(j)$ $j \in [1:4]$

For $C \in C_{values}$:

For j from 1 to 4:

We fit the logistic regression model using 3 subsets as a training set and we obtain an estimate for β by using a gradient method

We test this classifier on $X_{train}(j), Y_{train}(j)$ *and we obtain a score* S_j

We then take the average of the four scores: $S_C = mean(S_j \ j \in [1:4])$

Return: $C_{best} = argmax(S_C)$

We then used the following algorithm to find the best value for *C*:

Input: N (number of iterations), X_{train}, Y_{train} *(the training set),* $C_{initial}$ *(The best parameters of the good area found before).*

Initialize $C_{res} = C_{initial}$

For j from 1 to N:

$C_{range} = [C_{res} * 10^{-\frac{1}{2^j}}, C_{res}, C_{res} * 10^{\frac{1}{2^j}}]$

We use the previous algorithm with X_{train}, Y_{train} *and* $C_{range} : C_{best}$

$C_{res} = C_{best}$

Return C_{res}

The figure 2.10 shows the scores obtained on the training and test sets depending on the number of iterations:

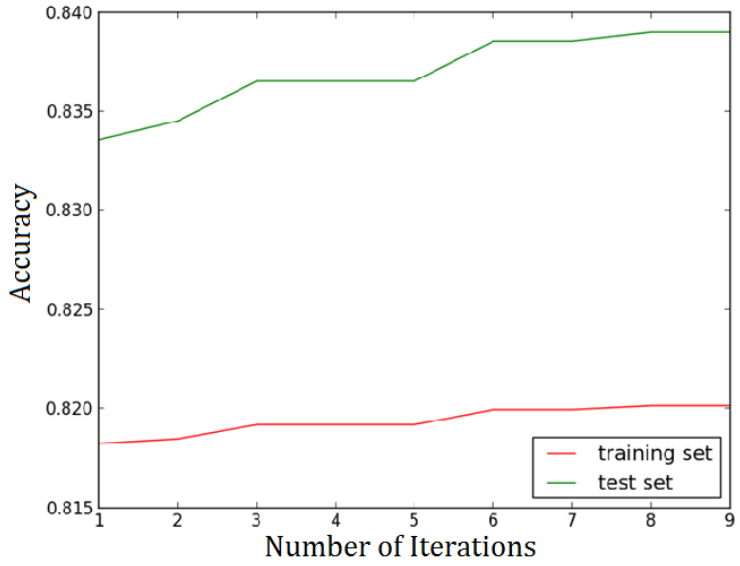

Figure 2.10: Accuracy of the classifier on the test set and on the training set for each iteration.

By iterating the algorithm, we managed to increase the accuracy score on the test set from 83.36% to 83.90% on the test set (Figure 2.10)

(ii) Significance of the model

We can first evaluate the global significance of the model by performing a likelihood-ratio test:

$$H_0: \beta_j = 0 \; for \; j = 1 \dots q$$

$$H_1: \exists \; j \; \in \{1, \dots, q\} tel \; que \; \beta_j \neq 0$$

The test statistic is:

$$T = 2 * (l(\beta_0, \beta) - l(\beta_0, 0)) \rightsquigarrow \chi^2(q)$$

We obtain $T = 1326{,}18$ so the model is globally highly significant. We can also test the significance for individual coefficients by performing a Wald test:

$$H_0: \beta_j = 0$$

$$H_1: \beta_j \neq 0$$

The test statistic is:

$$W = \widehat{\beta_j} / \hat{V}(\widehat{\beta_j}) \rightsquigarrow \chi^2(1)$$

For example, we have W(107)=11,23 for and W(2195)=13,87 for β_{107} and β_{2195} so both are significant.

(i) Marginal effect of a feature

The matrix X contains the features of the messages. The coefficient *(i,j)* of this matrix is the **term frequency–inverse document frequency** of the word *j* in the message *i*. The tf-idf is a measure of the importance of a word in a message. The tf-idf value increases proportionally to the number of times a word appears in the message and it is controlled by its frequency in a large number of documents: it balances the fact that some words are generally more common than others. Logistic regression allows us to find the feature which has the largest impact on the classifier's decision on a message of the test set. We indeed have quantitative variables and we can estimate the marginal effect of the raise of 1 unit of the value of a feature on $P(X)$.

$$\frac{\partial P(X)}{\partial x_j} = F'\left(\beta_0 + \; {}^tX\beta\right) * \beta_j = \frac{exp\left(-\left(\beta_0 + \; {}^tX\beta\right)\right)}{(1 + exp\left(-\left(\beta_0 + \; {}^tX\beta\right)\right))^2} * \beta_j$$

This effects depends on the value of X_i : we can estimate the average effect on all messages of an increase of one unit of the value of a feature $E[\frac{\exp\left(-\left(\beta_0 + \; {}^tX\beta\right)\right)}{(1+\exp\left(-\left(\beta_0 + \; {}^tX\beta\right)\right))^2}] * \beta_j$.

Let's have a look at the values for the features which have the largest estimates for β_j : $\widehat{\beta_{107}} = -4{,}27$ and $\widehat{\beta_{2195}} = 5{,}94$.

$$E\left[\frac{exp\left(-(\beta_0 + {}^tX\beta)\right)}{(1+exp\left(-(\beta_0 + {}^tX\beta)\right))^2}\right] * \beta_{107} = 0.040 * (-4,27) = -0.171$$

$$E\left[\frac{exp\left(-(\beta_0 + {}^tX\beta)\right)}{(1+exp\left(-(\beta_0 + {}^tX\beta)\right))^2}\right] * \beta_{2195} = 0.040 * 5,94 = 0.239$$

This means that if the tf-idf of the *"word 107"* increases of *1%*, the probability for an average message to be an insult decreases of *0.0017.* This word may be a nice word such as "nice", "sweet " or "kind".

However a *1%* increase of the tf-idf of the "*word 2195"* increases the probability for an average message to be an insult of *0.00239.* This word may be an insult itself such as "fuck".

Here are the principal results of this part (Table 2.3):

	Accuracy	Recall Rate
SVM algorithm without weights	84%	57%
SVM algorithm with weights	82%	71%
Logistic Regression	84%	57%
Decision Tress	83%	50%

Legend: *Recall Rate* $\geq 70\%$, *Recall Rate* $\geq 50\%$ *Recall Rate* $< 50\%$

Table 2.3: Misclassification Rate and Recall Rate of the classifiers of II) on the test set.

So far, our best classifier is the SVM algorithm with weights because his misclassification rate is really similar to the others, but his recall rate is really better. Detecting 71% of the insults while keeping a good overall accuracy (82%) is a quite good point for us, indeed we are tracking insults, so the more recall rate is high, the better it is. Even if the results can be considered as good enough, we wanted to "boost" our classifiers thanks to well-known methods like AdaBoost. Indeed, one of the major developments in machine learning in the past decade is the Ensemble method, which finds a highly accurate classifier by combining many moderately accurate component classifiers. We will principally study Adaboost with SVM-based and Decision Tree based component classifiers and Forests of randomized trees.

III. ENSEMBLE METHODS

A) Forests of randomized trees

- **Random Forest:**

As for every ensemble method, the goal of forests of decision trees is to combine the predictions of several different models built with a given learning algorithm in order to improve the robustness and generalizability of a single model.

In random forests, an arbitrary number of decision trees are created and combined, and each of them is built from a bootstrap sample (a random vector sampled independently and with the same distribution for all trees in the forest) from the training set (Figure 3.1). Furthermore, an arbitrary number "n" is set and at each node, during the construction of the trees, n predictor variables are randomly selected from all the predictor variables. And the split that is chosen is the best split among that random subset of all the features. Depending upon n, there are three different systems, including the random forest for values strictly between 1 and q, with q the total number of features. The inventor of random forests recommended three distinct values for the best results: $\frac{1}{2}\sqrt{q}$, $2\sqrt{q}$ and $\sqrt{q}$ –which is the default value in scikit-learn.

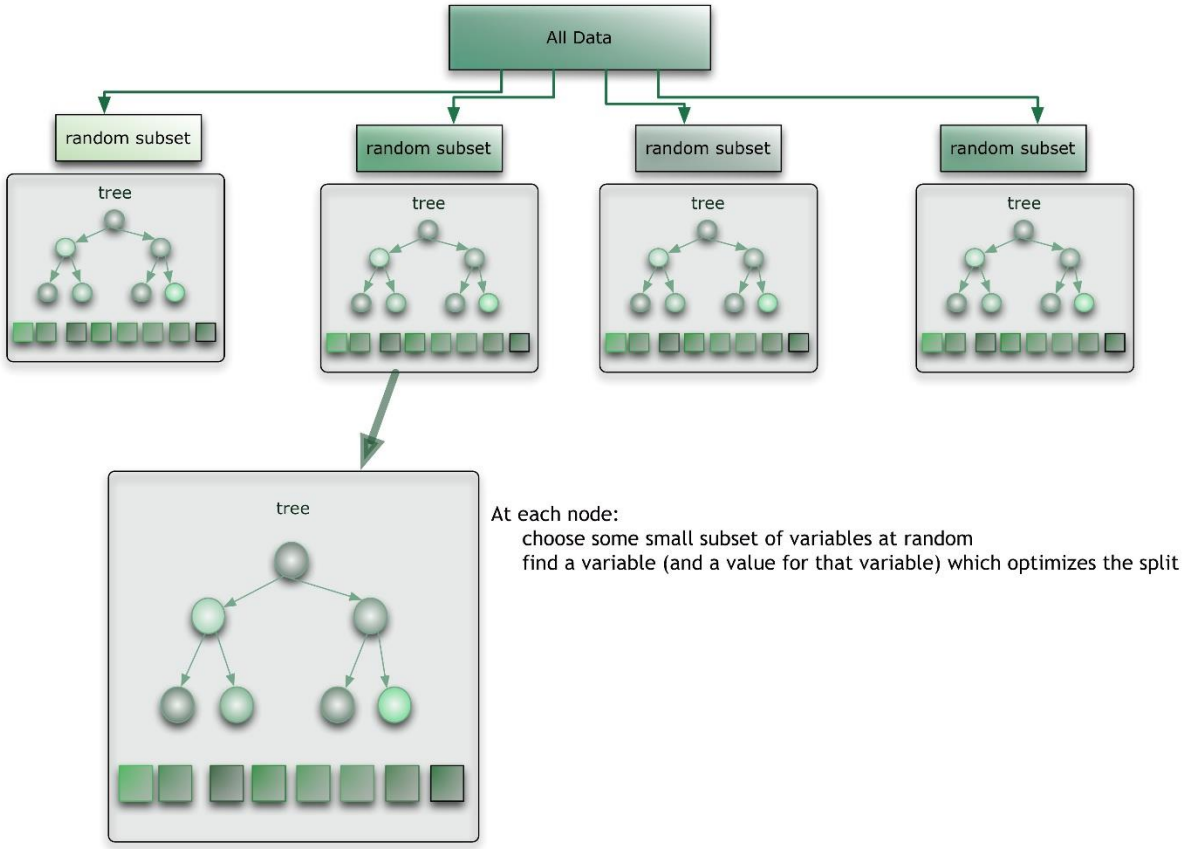

Figure 3.1: The creation of a random forest (adapted from http://citizennet.com/blog/2012/11/10/random-forests-ensembles-and-performance-metrics/).

As a result of this randomness, the bias of the forest usually slightly increases and variance decreases, due to averaging. The loss of variance usually more than compensate for the increase in bias, hence yielding an overall better model.

- **Extremely randomized trees**

The algorithm of growing extremely randomized trees is similar to the one growing random forests with two essential differences:

(i) A random subset of candidate features is used, as in random forests, but instead of looking for the most discriminative threshold, thresholds are drawn at random for each candidate feature and the best of these randomly-generated thresholds is picked as the splitting rule.

(ii) Extremely randomized trees don't apply the bootstrap aggregating procedure to construct a set of the training samples for each tree, so that the same input training set is used to train all trees.

As randomness goes one step further in the way splits are computed, the variance of the model usually decreases a bit more than for the random forests, at the expense of a slightly greater increase in bias (Figure 3.2).

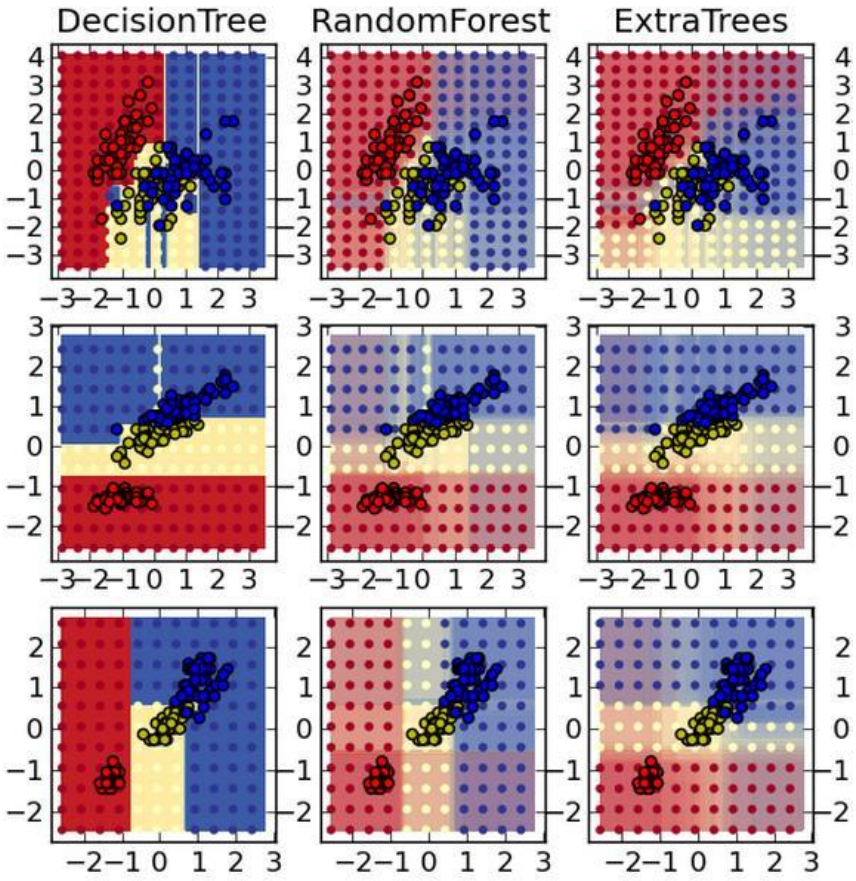

Figure 3.2: Decision tree, random forest and extremely randomized trees' decision surfaces on feature subsets (adapted from http://scikit-learn.org/stable/supervised_learning.html).

In addition to providing almost the greatest accuracy rate of the various algorithm used for in this paper, random forest and extremely randomized trees runtimes are rather fast, and they are able to deal with imbalanced and missing data.

Simulations:

Here is a graphic showing how the accuracy changes depending on the number of trees in the random forests and in the extremely randomized trees forests:

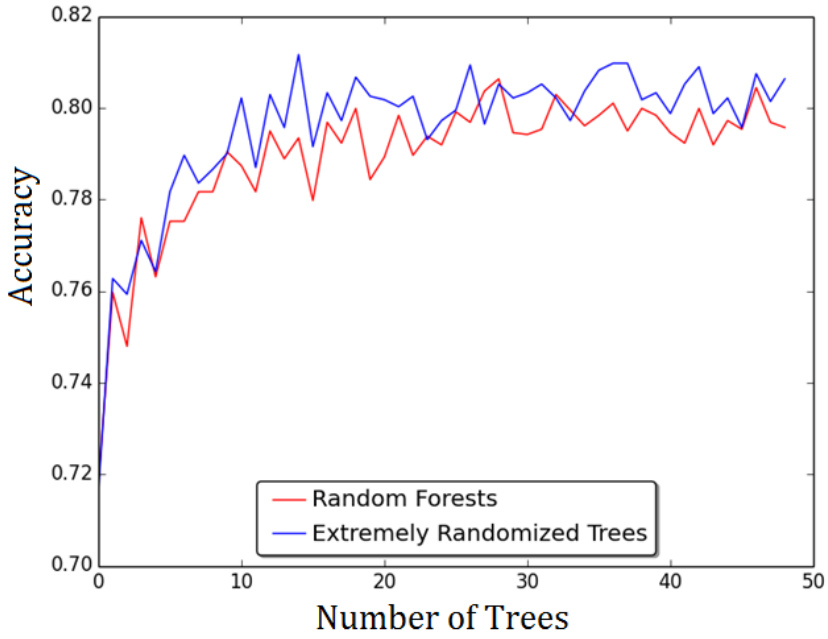

Figure 3.3: Precision rate of random forests and extremely randomized trees.

We went from an overall accuracy of about 72% to around 80% with the random forest and the extremely randomized trees algorithms (Figure 3.3). It could still be improved by trying out different values for each of the algorithm parameters and especially the maximum number of features randomly selected when splitting the dataset at a node, or by balancing the training dataset. And here is the graphic showing how the recall rate evolves depending on the number of trees in the random forests and in the extremely randomized trees forests:

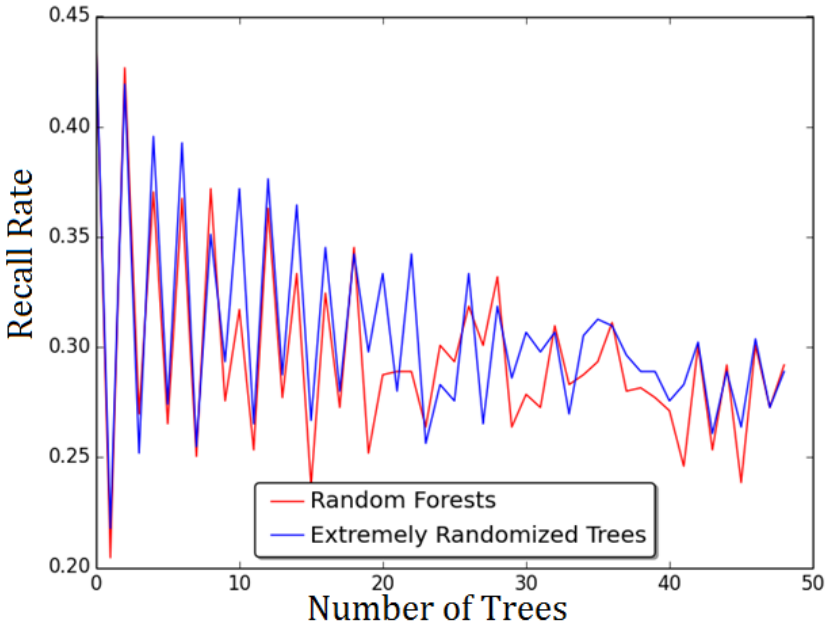

Figure 3.4: Recall rate of random forests and extremely randomized trees.

As you can see, the recall rates of these algorithms are really low and are not really improving over the number of trees created (Figure 3.4). In order to improve that recall rate, we can accentuate the prominence of the insulting messages in comparison with the other messages. The following graphics show the precision and recall rates of the random forests and extremely randomized trees algorithms with sample weights. The samples were weighted by class. The "normal" messages were given a weight of 0.001 whereas the insulting messages carried a weight of 1.001.

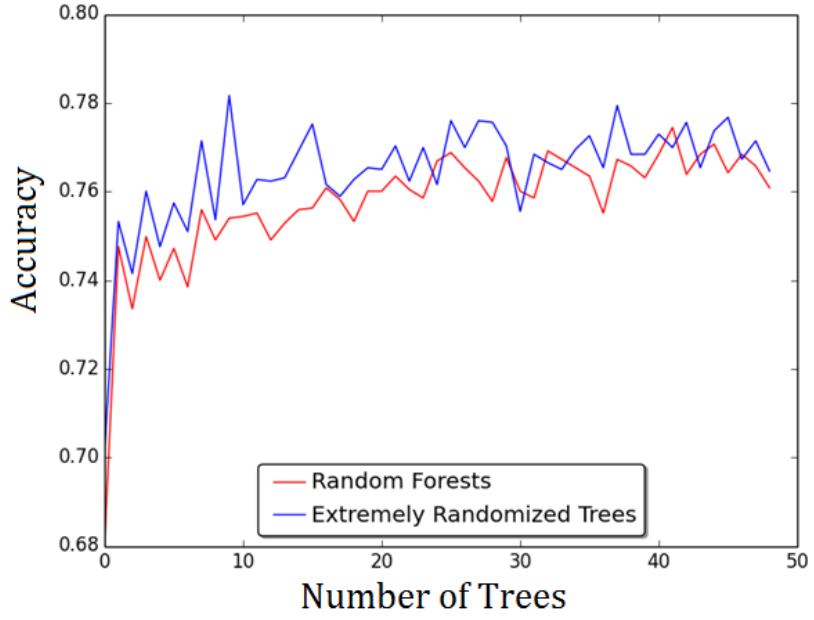

Figure 3.5: Precision rate of random forests and extremely randomized trees with sample weights.

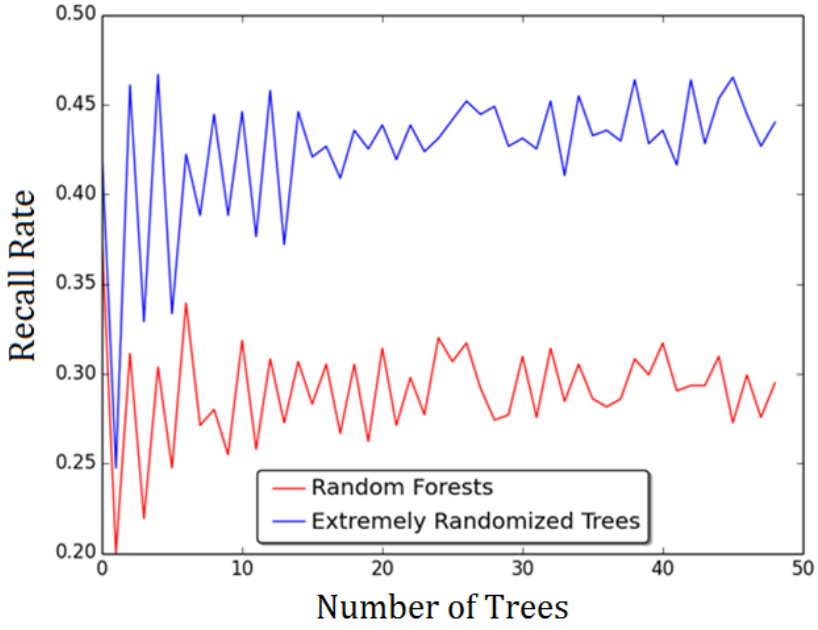

Figure 3.6: Recall rate of random forests and extremely randomized trees with sample weights.

As expected, the precision rate of both algorithms decreased, going from 80% to around 76% (Figure 3.5). Also the recall rate of the random forest algorithm does not really improve, while the recall rate of the extremely randomized trees algorithm goes up from 30% to more than 40% (Figure 3.5). The recall rates are still very low, for this reason we will now focus on another ensemble method that gives us better results for the error of the first kind: the AdaBoost algorithm.

3) Adaboost

- **Adaboost with SVM-based component classifiers**

Theory:

Given a training data set, Adaboost maintains a weight distribution, *W,* over the data points. Then Adaboost calls a classifier (which is called the ComponentLearn : SVM here) repeatedly in a series of cycles. At a cycle *t,* Adaboost provides the training data, with its distribution W_t over its data points to the ComponentLearn. In response, the ComponentLearn trains a classifier h_t(SVM with fixed parameters γ and C, the distribution W_t is then updated thanks to what the classifier h_t learnt (how well did it classify the training samples). Here we want to see if it's possible to use **RBFSVM** (SVM with Radial Basis Kernel Function) as a training classifier at each step.

We admit (from a research paper, reference [6]) that γ is a more important parameter compared to C: although RBFSVM cannot learn well when a very low value of C is used, its performance largely depends on the γ value if a roughly suitable C is given. This means that, over a range of suitable C, the performance of RBFSVM can be changed by simply adjusting the value of γ.

An easy way is to simply apply a single γ to all RBFSVM component classifiers. However, thanks to the reference [6], this way cannot lead to successful AdaBoost due to the over-weak or over-strong RBFSVM component classifiers encountered in Boosting process. We used a way that has already been used to use SVM with Adaboost. The process is to set an initial value of γ which is large (corresponds to a very weak classifier), and then to observe if its accuracy is above 50% or not, if it is the case we update the distribution W_t thanks to the output of the classifier, else we slightly decrease γ. Here is the pseudo-code of the algorithm:

Algorithm:

Input: Training data set: $\{(\vec{x}_l, y_l), 1 \leq l \leq m, \forall l\ y_l \in \{-1,1\}, \vec{x}_l \in \mathbb{R}^q\}$, $\gamma_{initial}$, γ_{step}, γ_{min}.

Step 0: Initialize W the weights of training sample: $W_0 = \left(\frac{1}{m}, \dots, \frac{1}{m}\right) = (w_0^1, \dots, w_0^m)$

Step 1: While $\gamma > \gamma_{min}$*:*

(i) Train RBFSVM component classifier h_t*, on the weighted training set*

(ii) Calculate weighted training error: $\varepsilon_t = \frac{\sum_{i=1}^{m} w_t^i * 1_{\{y_i \neq h_t(\vec{x}_i)\}}}{\sum_{i=1}^{m} w_t^i}$.

If $\varepsilon_t > 0.5$:

Decrease γ *from* γ_{step} *go to (i).*

Else:

Set the weight of component classifier h_t, $\alpha_t = \frac{1}{2} * ln(\frac{1-\varepsilon_t}{\varepsilon_t})$

Update the weights of training samples: $w_{t+1}^i = w_t^i * exp(\alpha_t 1_{y_t \neq h_t(\vec{x}_i)}))$ *(Note that the weight of well-classified data points is not changed)*

Output: A classifier f: $1 \leq i \leq m, f(\vec{x}_i) = sgn(\sum_{t=1}^{T} \alpha_t h_t(\vec{x}_i))$

Simulations:

We first tried to use SVM without weights as ComponentLearn but the first kind error was really high because the lack of accuracy of each classifier was not a good match with our imbalanced data set, indeed every message was said as non-insulting nearly all the time. So we decided to change a little bit the algorithm using Weighted SVM as component classifier, we first used the same weights than in part II)-A). We also tested smaller weights for insulting messages class because they are more often misclassified, and it's already taken care of in Adaboost algorithm. Eventually Adaboost with SVM based component classifiers enabled us to improve the recall rate by 15% compared to SVM with weights. Nevertheless, the accuracy decreased by 9%, this is due to an increasing of the second kind error (clean messages classified as insulting messages). For other simulations (by changing $\gamma_{step}, \gamma_{min}$ and the weights of the SVM), we also obtained good results (Table 3.1).

	Accuracy	Recall Rate
Simulation 1	73%	86%
Simulation 2	77%	79%
Simulation 3	76%	81%

Legend: *Recall Rate* $\geq 70\%$, *Recall Rate* $\geq 50\%$ *Recall Rate* $< 50\%$

Table 3.1: Misclassification Rate and Recall Rate of the AdaBoost classifier on the test set.

- **Adaboost with Decision Tree based component classifiers**

The algorithm is the same than in part III)-B), but the component classifiers are decision trees.

Simulations:

The results are not as good as the first part of III)-B) when using decision trees as ComponentLearn. This is probably due to the fact that SVM are more efficient than decision trees in our problem. Adaboost with Decision tree based component classifier returned an accuracy equal to 19% and a recall rate equal to 56%. However it is still better than using decision trees alone (see Table 2.2). Recall rate and accuracy increased by 5%. Here are the principal results of this part:

	Accuracy	Recall Rate
Random Forests	87%	30%
Extremely Randomized Trees	87%	45%
AdaBoost with SVM-based component classifier	83%	86%
AdaBoost with Decision Trees based component classifier	81%	56%

Legend: *Recall Rate* $\geq 70\%$, *Recall Rate* $\geq 50\%$ *Recall Rate* $< 50\%$

Table 3.2: Misclassification Rate and Recall Rate of the classifiers of III) on the test set.

CONCLUSION

Machine learning algorithms have participated to the development of automated text categorization. It is also used by several companies for many reasons:

- These classifiers can adapt themselves to any classification problem, just by choosing your classes and explanatory variables, so the application domains are endless.
- As the amount of data is increasing, information that can be pulled off becomes more and more important, and it implies to have more automated decisions without human judgment.
- It also has an impact on human classifiers, in fields where decision cannot be taken without a final human judgment. For instance, our classifiers can reduce in an important way the number of messages you have to analyze, if you are tracking insults or if you want to know what is said about your product on Twitter for example. (More and more companies are initiating projects around social media that involve automated text categorization).
- The levels of effectiveness of machine learning text classifiers have reached levels comparable to those of manual text categorization experts. Plus, manual text categorization is unlikely to be improved by the progress of research while effectiveness levels of automated text categorization keep growing. Even if they are likely to stagnate at a level below 100%, this level would probably be better than the effectiveness levels of manual text categorization
- The levels of effectiveness of machine learning text classifiers have reached levels comparable to those of manual text categorization experts. Plus, manual text categorization is unlikely to be improved by the progress of research while effectiveness levels of automated text categorization keep growing. Even if they are likely to stagnate at a level below 100%, this level would probably be better than the effectiveness levels of manual text categorization

Our work used three of the most well-known classifiers: Support Vector Machines, Decision Trees and Logistic Regression. We also tried to improve these algorithms by modifying them and by using ensemble methods and we had good results (Table 4.1). In our case, we found that methods that sum evidence from many or all features (SVM) tend to work better than ones that try to isolate just a few relevant features (decision-tree). The **AdaBoost with SVM-based component classifier** is our best classifier, we recommend it for text categorization.

	Accuracy	Recall Rate
SVM without weights	84%	57%
SVM with weights	82%	71%
Decision Trees	84%	57%
Logistic Regression	83%	50%
Random Forests	87%	30%
Extremely Randomized Trees	87%	45%
AdaBoost with SVM-based component classifier	73%	86%
AdaBoost with Decision Trees based component classifier	81%	56%

Legend: *Recall Rate* $\geq 70\%$, *Recall Rate* $\geq 50\%$ *Recall Rate* $< 50\%$

Table 4.1: Misclassification Rate and Recall Rate of our classifiers on the test set.

References

[1] Francis Bach. Stochastic gradient methods for machine learning. In Big Data related conference at ENS Paris, 2013.

[2] Andrew Ng. Supervised Learning. In Stanford machine learning lectures.

[3] Léon Bottou & Olivier Bousquer. The Tradeoffs of Large Scale Learning.

[4] Aditya Krishna Menon. Large-Scale Support Vector Machines: Algorithms and Theory. In Research Exam, University of California, San Diego, 2009.

[5] Akbani R., Kwek S. & Japkowicz N. Applying Support Vector Machines to Imbalanced Data Sets.

[6] Li X., Wang L. & Sung E. Adaboost with SVM-based component classifiers.

[7] Devroye L., Gyorfi L. & Lugosi G. A Probabilistic Theory of Pattern Recognition.

[8] Scikit-Learn documentation: http://scikit-learn.org/stable/index.html

[9] Breiman L. Random Forests, 2001.

[10] Chen C., Liaw A. & Breiman L. Using Random Forest to Learn Imbalanced Data, 2004.

[11] Geurts P., Ernst L. & Wehenkel L. Extremely randomized trees, 2006.

[12] Mee Young Park, Trevor Hastie Penalized Logistic Regression for Detecting Gene Interactions.

[13] http://www.stat.cmu.edu/~cshalizi/uADA/12/lectures/ch12.pdf a course about logistic regression.

[14] Arnak Dalalyan, Cours d'Apprentissage et Data Mining, ENSAE ParisTech.

[15] A gentle introduction to random forests: http://citizennet.com/blog/2012/11/10/random-forests-ensembles-and-performance-metrics/

[16] Hervé Frezza-Buet, Machine à vecteurs supports, Supélec.