The Mærsk Mc-Kinney Møller Institute
University of Southern Denmark
Odense, Denmark

Facultat de Informàtica de Barcelona
Universitat Politècnica de Catalunya
Barcelona, Spain

Master's Thesis – Projecte Final de Carrera

Support Vector Machines

Similarity functions to work with heterogeneous data
and classifying documents

Juan Manuel Parrilla Gutiérrez

Supervisor:
John Hallam (SDU)

Co director: Nasrullah Memon (SDU)
Co director: Enrique Romero Merino (FIB-UPC)
Project Period: Spring 2010

Juan Manuel Parrilla Gutiérrez (280485) – jupar09@student.sdu.dk

Preface

This thesis document was done in Denmark although I am an student from FIB-UPC Spain, as part of the program Erasmus. It will be evaluated in Denmark and the credits will be approved in Spain. There this thesis is called “Final Project” and once it is passed I will have completed “Informatics Engineering”, 5 years studies which are considered to be like bachelor + master. That is why this thesis in Denmark is considered a Master's Thesis although I am not a Master student.

This document was done using “OpenOffice.org” because I lack the minimum knowledge to do it in Latex. I promise next time I will have to write a technical document I will use Latex. English is not my native language that is why I tried to use simple sentences and expressions.

This document contains three major chapters:

The first one is the Introduction, there I explain all I learned about SVM and how we will apply our idea.

The second one, called “middle chapter” is about using a Data Mining tool and a Natural Language Processing tool to classify documents.

The third one, called “last chapter” is about integrating the ideas described in the introduction in a Data Mining tool.

Each chapter contains its own conclusions, however there is a final chapter where I talk about what I did manage to finish and what I did not.

Acknowledgments

To Enrique Romero. This project is his idea. He waited for me during 3 years until I arrived to my Final Project. He let me do this project abroad and it was not a problem for him if he could not be a supervisor. He taught me all I know about SVM and while I was doing this project he answered to several e-mails. We discussed everything and he supervised every step I did.

To John Hallam. He let me do my project here and he let me work in one of his computers at MMMI offices.

To Soeren Sonnenburg. He answered all my questions about Shogun, without him I would be stuck in the very beginning. He is a model about how to maintain an open source project.

To Michael Witten. He helped me developing C++ and fixed my biggest bug.

To the #C++ and #Python channel in IRC's Freenode. They helped me with both languages.

Abstract

The objective of Data Mining (DM) is to classify information from the real world. That kind of information is commonly heterogeneous data: information that needs different kind of data to be represented. How to deal with heterogeneous data has been usually something DM lacks about because DM is not deeply used with real world problems. Different solutions has been shown and our objective is to show a new one using similarities and Support Vector Machines (SVM). How to use similarities instead of kernels in SVM and later how to combine similarities to work with heterogeneous data. The idea is that any type of data will have a similarity related and then all this similarities will be combined to output a result. What makes this idea powerful is the way we can combine similarities, it can be practically anything while other methods to work with heterogeneous data only do linear combinations.

Our Goals

First of all understand how SVM works and what does it means to use similarities instead of Kernels. Later implement in a SVM library what explained before and show it working with an example. We will work with documents so it would be also required to do some NLP, learn about a NLP is another of my goals.

Another of our goals is to use OO techniques and get a good design. Make our framework easy to be modified by anybody. Make an easy implementation. The objective is to extend the library used not to fork it.

Index

1.Motivation 1
2. Introduction 2
  2.1 Support Vector Machines 3
  2.2 Kernel 6
  2.3 Soft Margin 8
  2.4 Similarity 9
3. Classifying documents 12
  3.1 Set up the framework 13
  3.2 From documents to real numbers 17
  3.3 Working with “Movie Reviews” corpus 17
    3.3.1 1st feature selection: frequency thresholding 18
    3.3.2 Removing non-alphanumeric and stopwords 21
    3.3.3 Normalizing words 23
    3.3.4 Feature selection: information gain 24
    3.3.5 How to represent a document 27
    3.3.6 Collecting new data 28
  3.4 Working with the “Brown Corpus” 30
    3.4.1 Improving TFIDF 31
  3.5 Chats 34
  3.6 More about documents representation 34
  3.7 Summary 35
4. Making texts similar 38
  4.1 Can we use any similarity with SVM? 40
  4.2 Why using a similarity instead of a kernel? 42
  4.3 What is heterogeneous data? 42
  4.4 Our Solution 44
  4.5 Other (new) solution 46
  4.6 Implementation 47
    4.6.1 Implementing similarities 47
    4.6.2 Similarities or kernels 53
    4.6.3 Making it work with Python 53
    4.6.4 Combining similarities 55
    4.6.5 Finally heterogeneous data 57
    4.6.6 Some extra work was required 60
    4.6.7 Complete design 61
  4.7 Comparing in with MKL 62
5. Conclusion 63
6. Further work 64
7. Bibliography 65
8. Our contribution 67
A. Appendix 68

1. Motivation

It all started 3 or 4 years ago. At my home university (FIB – UPC Spain) with the old degree (the one I am finishing now, nowadays they are changing everything) there were four courses that were projects. Most of the courses had a project related and we also had to do what is called “final project” (now thesis), but there were four courses that were just a project, a big one actually, it stole part of our lives: one about the net, one about operative systems (do one), one about software engineering and one about programming. At the same time the students were supposed to do other 3 o 4 courses. The only useful thing I can see about that was make us work under big pressure for lots of hours in a row. It is common for a computer guy to work around 10-12 hours everyday, 6 days a week (or more) in Spain, although the law of courses says that the maximum is 40. So we were kind of prepared for the “real world”.

The important course in this story is the one called “Programming project”. It is the first big project you usually do. The objective of that course is to make you write a lot of code, dozens of classes, write as mad. Then you go to class after one week working on it everyday several hours and the teacher makes you change everything. You also lack a lot of information, usually when doing ProP no one has done any software engineering course and they require it, they also require AI knowledge while at that moment you don't have any idea about it. That project is done in groups of 3 people and there are 3 projects that must be merged in one application, in total 9 people. It can be seen as a “Big Brother” experiment.

I was very lucky about the topic I did. Usually the projects are about creating a system to control an airport or something like that. I was very lucky because I had to do a Neural Net, from zero. I had to learn how they worked and implement one, we used Back Propagation. At that point I never did anything related with DM or AI. I really enjoyed that and I searched for better ways to improve my Neural Net and I tried to implement RProp and QuickProp. Sadly I did not manage to get them working (it was enough with Back Propagation to pass it). So my teacher pointed me to Enrique Romero to try to fix it.

We met on summer time when the classes were done. It was for fun. This is when he offered me to do my “final project” about Neural Nets, then as explained in the last big chapter that idea was moved to SVM. We met more or less and he taught me most I know about DM, NN and SVM. It was supposed that we were going to do the project together but I decided to do it abroad and Enrique let me do his project in another university. Luckily it was OK for John Hallam to do that project about SVM.

That is why I am doing this project. That is when probably my love for AI and DM started although with the years passing by and getting a better perspective I see that I like the fields were I had good teachers.

That is my *personal* motivation. This is the last chapter I am writing and I explained several times (in the introduction and in the last big chapter) the real motivation so I feel like I am going to make lost time to the reader. Briefly, the objective is to show a better way to work in Data Mining with heterogeneous data.

2. Introduction

Support Vector Machines (SVM) [1] are an “state of the art” Data Mining tool for different reasons. SVM usually offer better results than other methods, they have no problem with local minima (the big issue with Neural Nets), SVM don't require to specify many parameters as other methods do, usually the capacity (explained later) and the Kernel to use (and any parameter required by the kernel). SVM can work very easily with thousands of different features, they are usually very fast and finally, what makes a big difference is that SVM uses a Kernel function. I will go a bit deeper later, but what makes a Kernel a very powerful idea is that we only have to think about the kind of data we are working with.

Provided we are working with a good Kernel suitable for the problem much of the preprocess usually done to the data is not required and what is more important is that the Kernel will let us work with the data in a “near raw state”. We can focus on how to classify the data in the same way we would do in real life. We can abstract ourselves from all the mathematic and statistics problems usually one have to deal when working on Data Mining and just concentrate on the data itself.

If we are experts about what we are working on, finding a function to classify it should be easy, and once we have that function construct a Kernel is straightforward (usually just apply it) and the problem is solved.

This thesis is about SVM and Kernels. At first I will work with a framework and classify documents doing all the preprocess outside the SVM and using standards Kernels like Gaussian or Chi2. In that stage I will focus more on the problem of classifying texts, extracting features, using different similarity function between documents and NLP. The objective is also to learn to work with Shogun SVM; the framework I will be using through all the thesis, first as an end user, later as a developer, and learn how to integrate a NLP (NLTK) library with a DM (Shogun) library. The language used will be Python.

The second stage is about exploring the concept of similarity and Kernel which I will explain later. The basic idea is, if a Kernel is computing how similar are two training points (usually done by a dot product being a training point a vector in some space), why not use a similarity function instead?

The big point about this simple idea is that it will let us work with heterogeneous data in a very easy and direct way without much preprocess while nowadays when working with heterogeneous data it has to be converted to (usually) real numbers before starting the classifying method.

The philosophy and “motto” about this thesis related with Data Mining can be defined in:

- Solve the problem as you would solve it if working with real world entities.
- The data itself is enough. Think about the data as it is, not about what mathematically/statistically it is.
- There is no need to make the problem *transparent* to the end user.

We think that since we solve real problem in our real world in a daily basis it should be easier if Data Mining problems can be solved in the same way. If we know how to classify something in real world the same solution must be valid in a DM context.

We provide a framework ("similarity" concept, explained later) and we show the results classifying documents.

2.1 Support Vector Machines

What follows is not an introduction about SVM. It is not my intention to go deep in theory or cover all the details, I will only cover in a quick way the key points about SVM that I may use later. Part of this thesis is about learning SVM (around three weeks). Usually thesis reports offer results about experiments, I will offer what I have learned as a result here. For a good introduction about SVM the common checkpoints are [2] [3] [4] [5]. What follows is learned from those texts.

If we have a two-class dataset what an SVM will do is finding an hyperplane that will divide the space where the data is. All the points at one side of the hyperplane will belong to one class and all the points at the other side will belong to the other class (multi class problems are usually done one vs all way, it can also solve multi-label problems [6]). That is how SVM classifies.

The equation of an hyperplane is defined as $\langle w \cdot x \rangle + b = 0$

Given this situation we can define two margins: functional and geometric. The functional margin is the value we obtain using in the hyperplane equation a training point where x is multiplied by his label, the functional margin of a (x,y) point is $y \cdot (\langle w \cdot x \rangle + b)$ where the value of y is -1 or +1. The geometric margin is the euclidean distance from any point to the hyperplane.

The objective of SVM is to find the critical points in each class set. The critical points will be those that are very near to the hyperplane so given the set of critical points and any other other point we can know to which class the other point belongs comparing it to the critical points.

The critical points are a boundary of the class set. We will focus on them and we will fix the functional margin to 1 or -1 (depending on the class) when applied to a critical point. Critical points are called Support Vectors. Support Vectors are all the information we will need to classify new points. All the other points will not be considered and that means that given a train set with thousands of entries we will only work with a very little set.

The critical points of each class are defining an hyperplane too, so we have two more hyperplanes with functional margin 1 and -1 (the equations of the new hyperplanes will be $\langle w \cdot x^{+} \rangle + b - 1 = 0$ and $\langle w \cdot x^{-} \rangle + b + 1 = 0$, the sign over the x marks the class). There are different generalization bounds, however the most common is to reduce the problem to minimizing the norm of the weight vector (w), by minimizing it what we are achieving is maximizing the geometric margin (Euclidean distance) the most we can given the functional margins of 1 and -1. So now we have an equation (the hyperplane) and a condition (minimizing w) and that defines the problem.

The basic formulation of the problem is what follows:

Given a linearly separable training sample $S=((x_{1,} y_1)...(x_l, y_l))$ where “l” is the size of the training set, (x_i, y_i) represents an entry in the training set, where x is the vector that represents that entry and y is its class (as suggested before when working with SVM x can be anything, a vector of reals, an image,...).

The formulation of the problem is:

$$\begin{aligned} &\textit{minimise}_{w,b} && \langle w \cdot w \rangle \\ &\textit{subject}\text{ to} && y_i(\langle w \cdot x_i \rangle + b) \geq 1, \\ & && i=1,\ldots,l \end{aligned}$$

What follows now is a sequence of mathematic tricks to make that formulation as easiest to calculate as possible, the objective is to transform the problem to a dual formulation where the solution can be found as a linear combination of the training points.

We mix the objective and the condition in one equation using Lagrangian multipliers, in this way we obtain the primal form:

$$L(w,b,\alpha)=\frac{1}{2}\langle w \cdot w \rangle - \sum_{i=1}^{l} \alpha_i [y_i(\langle w \cdot x_i \rangle + b) - 1]$$

α are the Lagrange multipliers. If we now differentiate with respect to w and b:

$$\frac{\partial L(w,b,\alpha)}{\partial w} \rightarrow \quad (w=\sum_{i=1}^{l} y_i \alpha_i x_i)$$

$$\frac{\partial L(w,b,\alpha)}{\partial b} \rightarrow \quad \sum_{i=1}^{l} y_i \alpha_i = 0$$

And resubstituting the relations obtained into the primal form, we obtain the dual form

$$\begin{aligned} &\textit{maximise} && W(\alpha)=\sum_{i=1}^{l} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{l} y_i y_j \alpha_i \alpha_j \langle x_i \cdot x_j \rangle \\ &\textit{subject}\text{ to} && \sum_{i=1}^{l} y_i \alpha_i = 0 \\ & && \alpha_i \geq 0, \ \ i=1,\ldots,l \end{aligned}$$

The most important thing about this formulation is that now the solution can be described as a linear combination of the training points. A training point is represented as a vector and we are performing an inner product between them; the biggest is the result of the inner product, the most both vectors are pointing to a similar direction, the most similar that two vectors (training points) are.

Now we use one of the key concepts used in SVM, the KKT conditions, the conditions state that the optimal solutions $\alpha^*, (w^*, b^*)$ must satisfy:

$$\alpha_i^*\left[y_i(\langle w^*\cdot x_i\rangle+b^*)-1\right]=0, \quad i=1,\ldots,l$$

This means that or α^* is 0 or $\left[y_i(\langle w^*\cdot x_i\rangle+b^*)-1\right]$ is 0 and we know that we forced what we called critical points to have a margin of 1 or -1 (the classes will be defined as 1 or -1), so we know that $\left[y_i(\langle w^*\cdot x_i\rangle+b^*)-1\right]$ will be zero in that case and then α^* will be non-zero. All other points will have $\left[y_i(\langle w^*\cdot x_i\rangle+b^*)-1\right]$ different from zero so α^* will be zero. And that is how we find the critical points called support vectors. Searching for that points with an α^* different from zero. And that is what SVM is about. At first I told that SVM search for an hyperplane that divides the data in classes.

Actually what SVM does is to search for that points called support vectors because that points are defining an hyperplane themselves, and that hyperplanes would divide the data in classes. Each hyperplane would divide the data in two classes: the one defined by the hyperplane and the rest. This is why at the beginning I said that multi-class problems are usually solved in a one vs all way.

Since the calculation of α is the most important in SVM and they came from a hard mathematic way I like to think about them in a simpler fashion.

The Perceptron Algorithm is a big loop that starts with random values as weights (or zero) and tries to classify the data. For each iteration misclassified training examples are added or subtracted from the weight vector until no mistakes are made or we arrive to an small error we think it is enough to stop the loop. Inside the loop we would have:

```
if $y_i(\langle w\cdot x_i\rangle+b_k)\leq 0$ then
    $w_{k+1}\leftarrow w_k+\eta\, y_i\, x_i$
```

If we start with the weight being zero the final weight will be a combination of the training points. This value of w is the same we get when differentiating with respect from w in the primal form. Looking at the dual form it is very similar to what we get in the function to maximize.

$$w=\sum_{i=1}^{l}\alpha_i\, y_i\, x_i$$

And here we have α again, now it means the number of misclassification x_i has caused, inside the loop we would have:

```
if $y_i(\sum_{j=1}^{l}\alpha_i y_j\langle x_j\cdot x_i\rangle+b)\leq 0$ then
        $\alpha_i\leftarrow\alpha_i+1$
```

The hardest points to classify will cause more misclassifications so they will have a bigger α value, and since what we are searching for are the critical points for each class, we are searching for the points with a bigger α . In SVM the idea is similar, though the search is easier because if a point has an α different from zero: it is a critical point.

2.2 Kernel

When working with SVM we are usually dealing with spaces with a very high dimensionality. For example in the tests I did about classifying documents and which results I will show later, I worked with spaces with around 1000 and 3000 dimensions. The objective of SVM is to find an hyperplane that separates the data in spaces we may don't know how to do it, or what it is worse, we may know that the data is not linear separable in the actual space (called input space).

Kernel is a tool that helps us in the sense that it projects the data from an input space to a feature space where we know that the data is linear separable (or where we know how to separate the data). What we do is using a function to transform the data from the input space to the feature space. We map the input space X to a new space, $F=\{\phi(x)|x\in X\}$, so returning to the dual form we can rewrite it as:

$$W(\alpha)=\sum_{i=1}^{l}\alpha_i-\frac{1}{2}\sum_{i,j=1}^{l}y_i y_j\alpha_i\alpha_j\langle\phi(x_i)\cdot\phi(x_j)\rangle$$

Suppose that we want to compute the similarity between documents, something that will be shown later. At first we have them in raw form, a sequence of symbols, we can't do the inner product between them and use the result as a similarity value. We are working in an space where we don't know how to separate the data (and maybe that space doesn't have an inner product).

What we can do is to define a new space, each dimension of that space will be represent a word that appears in one of the documents (or more) so it will have as many dimensions as different words. We will define each document as a vector in that space, where each $position_i$ of the vector represents the $dimension_i$, that position will be 1 if the document contains that word, 0 otherwise.

If we have 3 documents as follows:

$Document_1$ = {“car”,”bike”,”plane”}
$Document_2$ = {“plane”,”train”,”car”}
$Document_3$ = {“train”,”bike”,”boat”}

We have 5 different words and we will define a 5D space where $dimension_1$ will represent “car”, $dimension_2$ will represent “bike”, $dimension_3$ will represent “plane”, $dimension_4$ will represent “train” and $dimension_5$ will represent “boat”.

Now we project each document from the input space to the feature space:

$Document_1$ = {1,1,1,0,0}
$Document_2$ = {1,0,1,1,0}
$Document_3$ = {0,1,0,1,1}

What we have done in all this process if to define a function ϕ that projects from an input space to a feature space. From documents in a raw format which we don't know if they are separable (at least in a easy way) to a new representation in a higher dimensional space (each document went from 3 entries to 5, although is not always necessary to user a higher dimensional space, it uses to do the separation easier) where we can know in a very easy way if they are separable or not.

If we follow what appears in the dual representation and apply the inner product, we get:

$$document_1 \cdot document_2 = 2$$
$$document_1 \cdot document_3 = 1$$
$$document_2 \cdot document_3 = 1$$

This means that $Document_1$ and $Document_2$ are the most similar and that the data is easily separable. This demonstration may seem trivial, but this idea can be powered as much as we want, we can define more complex ϕ to project the input documents or given any kind of data and a way to represent it, we can use it as a ϕ. For example, when working with a set of images, we can focus on the edges and contrasts of colors and create a new space with that information.

The process is being done in two steps, project the data (use ϕ), calculate the inner product (the “similarity”) but if we know that the inner product is also available in the feature space we can mix both steps in one to build a non-linear learning machine. We call this mix a Kernel.

$$K(x,z) = \langle \phi(x) \cdot \phi(z) \rangle$$

The dual formula is rewritten as (and what follows may be called the basic proposition of the SVM problem):

$$\begin{aligned} &\textit{maximise} \quad W(\alpha) = \sum_{i=1}^{l} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{l} y_i y_j \alpha_i \alpha_j K(x_i, x_j) \\ &\textit{subject}\text{ to} \quad \sum_{i=1}^{l} y_i \alpha_i = 0 \\ &\qquad\qquad \alpha_i \geq 0, \quad i = 1, \ldots, l \end{aligned}$$

I can't avoid talking about Mercer's Condition now, however I will not go any deep with it. Mercer's Condition defines when this process may be done and when not. If the Kernel we are getting is semi definite positive it will work, otherwise it may work. It may not be always possible to check if our Kernel satisfies Mercer's Conditions.

In the first stage I will only use Kernels to compute the inner product, I will use different kinds of Kernels, like Gaussian, Chi2, Polynominal,... but all the projection from the input space to the feature space will be done outside the Kernel. I will send to the kernel a list of real numbers so the real power of kernels will not be used.

Sadly that is what is happening in most of the cases since it is easier to do the projection outside instead of defining a new Kernel (they are use to be considered something transparent to the user that classifies the data), but once a Kernel is constructed it is keeping a concept itself, not only a formula. It has become a tool and it can be reused easier that repeating all the projection each time outside the kernel.

Also, we can trace what is happening, we can see the data flowing, we can determine which ones are the support vectors and why (doing all the projection outside we only get vectors with real numbers as output). We can consider SVM something with can work with, not a transparent tool that offers good results.

2.3 Soft Margin

Until now I have talked about drawing a perfect line that perfectly separates the data, if it is not possible or we don't know if it is separable we can use a Kernel that will do it perfect anyway. That's not the case in real world scenarios. Although the data might be easy to separate with a line we may get points out of its class. With everything talked before that points would screw our problem. That is why soft margins were created, to let some points be out of class. The key here is a variable called C. Since we will mostly work with real world data the most used SVM libraries require to specify always a value for C.

The basic idea about C is, the bigger C the harder margin is, while the lower C the softer margin is. Until now we have not worked with C so the margin was very hard. Usually when getting similar results it is better to choose a lower C, it prevents overfitting, but that totally depends on the data.

When the current margin is violated by a training point we have to consider if that training point is out of class of it is defining a new margin. We call the distance from the current margin to the actual training point "slack", represented by ξ .We can rewrite the first formulation of the problem to let the margin be violated using slack variables, we are relaxing the constraints only when necessary, we are introducing a new cost:

$$\begin{array}{ll} \mathit{minimise}_{w,b} & \langle w \cdot w \rangle \\ \mathit{subject}\text{ to} & y_i(\langle w \cdot x_i \rangle + b) \geq 1 - \xi_i, \\ & \xi_i \geq 0, i = 1, \ldots, l \end{array}$$

When we have an error it means that the corresponding slack variable exceeds the unity, so $\sum_i \xi_i$ is an upper bound on the number of training errors. We can use this idea an assign an extra cost for errors in the objective function to be minimized, C means how big the penalization per error will be.

$$\begin{array}{ll} \mathit{minimise}_{w,b} & \langle w \cdot w \rangle + C \sum_{i=1}^{l} \xi^2 \\ \mathit{subject}\text{ to} & y_i(\langle w \cdot x_i \rangle + b) \geq 1 - \xi_i, \\ & \xi_i \geq 0, i = 1, \ldots, l \end{array}$$

Now follows some mathematic tricks to represent it in the dual form but this brief introduction should be enough to know how to play with the C parameter.

2.4 Similarity

The more far we are from mathematics and statistics and the more near we are to our data, the best results we will obtain. Usually Data Mining doesn't care about the kind of data, it has a very big arsenal of tools to extract all the info it cans. Usually Data Mining abstracts the problem to formulas, SVM let us to abstract the problem to the data and that is one of the reasons it is nowadays considered state of the art. If we are an expert in the problem (the word expert is widely used in Artificial Intelligence, and Data Mining can be probably considered part of it) we will get very good results and maybe most important, we will know why and in case of problems it will be easier to fix them. In the tests that will follow this introduction to SVM I didn't do much statistical analysis, I played with the words as would be done in real world.

As said before a Kernel does two steps, computes the projection to the feature space and calculates the inner product; the inner product seeks how much two vectors are pointing to the same direction, the most they do the more similar they are, why not working with a similarity function directly?

The objective of Data Mining is to classify real world problems and for doing that there is a “problem”: most real world problems contains heterogeneous data. Heterogeneous data is defined as a data set that contains different types of data, and all of them are required to get results. For example, if we are trying to guess what's wrong with a medical patient we will have to work with different data to predict the sickness, like a blood analysis, x-rays images, blood pressure... and only using all the information we can from all sources we can get a good output.

Another example is a robot moving in a map, the robot may have sensors to receive information from the world and analyzing that information predict (classify or use regression) the next movement. The information the robot is receiving can be an image of what the robot “sees”, a sound wave about what the robot “hears”, daytime to predict the light of the scene and interpret the image, wind velocity to interpret the sound wave to remove the noise, the path of other robots moving around, and so on.

We have different types of data and we have to get an output. Until now we have talked about one kind of data and one function to project it. Nowadays the way this problem is solved in SVM consists in a combination of Kernels, one Kernel for each kind of data, this means that we are adding a new parameter to estimate the weight assigned to each kernel (now usually called sub-kernel and using $k_i(x,y)$ instead of $K(x,y)$ [7].

The new Kernel, called combined Kernel, is calculated as:

$$k_{combined}(x,y)=\sum_{m=1}^{M}\beta_m k_m(x,y)$$

Where β is the new parameter we have to estimate. Adding this new parameter means that we have to modify our SVM algorithm, the new algorithm is called MKL (from Multiple Kernel Learning), it is a variation of the algorithm explained before, it is deeply explained in [7].

The idea of combining kernels lets us to work with heterogeneous data, but it is not using the information the data has itself. It is in a simple way, combining data and finding critical features.

Let's return to the examples I talked before, we have a medical patient with an x-ray image and a blood analysis, we want to know if the patient is sick or not. MKL will combine that two types of data and find relevant features, like “focus on this part of the image and these parameters of the blood analysis to tell if the patient is sick or not, the image is three times more important”.

What we suggest is a similarity function that is a simpler and more powerful way to combine heterogeneous data. Let's suppose that we have two training points, each one consisting in a x-ray image and a blood analysis. We want to know how similar they are, the objective of the problem is knowing which patients are sick and which aren't, so as always we can find the critical patients and test new ones against them.

Our similarity function can perform actions like: “if both entries have a similar value in some parameters about the blood analysis, check some part of the x-ray image”, “if the value of one training point the parameter x at blood analysis is above some threshold, but the other training point has that parameter under the threshold, check for other parameter”, “if some part of the x-ray is similar, don't check the blood analysis”.

Another example in criminal context is to classify faces with images and descriptions. Usually a witness is required to describe the face of a criminal, the witness would usually use words, the policeman would try to get an sketch. This information will be later checked against a database to search for suspect people. Suppose we have both types of data in our system. The system can “read” the description and focus on different parts in the pictures. If the description talks about hair the system would focus on the top part, if it talks about the nose, the system will check the middle part. If it doesn't talk about the mouth, the system would ignore it. If two pictures have similar eyes we can check the text descriptions and so on...

Also, another interesting side-effect is that if we lack some information we can still work and solve the problem (although the result will probably not be the best, we can still get a result). For example in a crime scene were there were not witnesses but we get a picture by a hidden camera we can still classify it. Missing values are more frequent in medical contexts: in emergencies doctors must take decisions with very basic information. We can define our similarity function to work when all the information is applied and we can also define it to work when some information is lacking, it is not just redo the linear combination, it can be a totally different way to solve it.

Provide we can compute a similarity function for each kind of data we can merge them in one function. What we may need is the advice from an expert. In the above scenario we will need a doctor to help us. We can ask the doctor, with an x-ray image and a blood analysis how do you know if a patient is sick or not? The doctor will explain to us in what he focuses or what he searches for, and is that kind of procedures what we will use in our similarity function.

If we don't have an expert we can combine the similarity functions in the same way MKL combines kernels. A kernel is a matrix so we can construct a set of matrices with the results from the similarity functions, one matrix for each similarity, give this information to MKL and it will find the best way to combine them.

Given all the SVM basic theory the only change we need to do is using a similarity function instead of a Kernel: each time we found K(x,y) replace it with S(x,y). And if we are lazy and want to still use the SVM libs already working, since a Kernel is a matrix we can construct a matrix filling it with the similarity values between all the pairs of training points. Use this matrix as a “Kernel” with our SVM algorithm and everything should work, some changes may be needed because the matrix should be semi definite positive, but it may work still.

Why using SVM and similarities? Because SVM are using in their calculation a concept very “similar” and because a similarity function is not necessary differentiable, and this means for example that we cannot use this idea in Neural Nets.

Working with similarity functions has another good side-effect: for some kind of data we can calculate how similar are n elements while kernels always compute it in pairs. As example, given 3 sequences of DNA we can say how similar they are by finding the common ancestor and counting the average number of changes from the initial sequence to the ancestor. That can be extended to n sequences. This is something that will not be explored in this thesis, but it would be nice to work with it because it can boost the execution of SVM.

3. Classifying documents

The objective in the introduction was about understanding how SVM works, the theory beyond them. The objective now is to make them work. We will use a library called Shogun [7], that library has several data mining tools; we will focus on classifiers, specially SVM. That library, among other feature types, wants to receive lists of real numbers with their labels as input, “anything” converted to real numbers would be classified by Shogun.

Since in software development one has to explain why some tools were chosen, we will also explain why we chose Shogun:

- Shogun is an alive project (compared to most of the others Data Mining projects), it is continually being updated[1].
- Shogun covers everything about SVM, it is in C++, a quick look at Shogun's code is enough to see that it is high quality done by expert people.
- Shogun's authors are top people in SVM nowadays, they are part of the MKL development.
- The community is alive, the authors are easy to contact and they provide quick help.
- Shogun is free software, they use a GPL v3 license. The code is high quality and this also means that is very easy to add new features (what we will do in the next chapter).
- The documentation is very good with dozens of examples, each class is very well explained, all the code is well documented (more than 80K lines). Develop to Shogun is easy.
- It is in the Debian repositories. That means an instant installation.

We will work with documents; the documents will be in a raw “machine readable” state, by that we mean that documents will be in what is known as “txt”, each byte will contain a character in (probably) ASCII format. That means that we will not cover the part where documents are converted (or parsed) from PDF, HTML, DOC,... although there are easy ways to do it and the Natural Language Processing (NLP) tool we will use support many of them.

To process the documents we will use a library called NLTK, there are lots of NLP libraries, why we chose NLTK?

- It is implemented in Python, Python is probably with Perl the best programming language to work with documents.
- Shogun works with Python, so if the NLP also works with Python everything will be easier.
- We found that the NLTK tutorial is very good and straightforward and they also have a book to cover it. [8]
- After a quick review, we found that NLTK did everything we need and also provide us with tools we didn't know before.
- The handicap is that Python is slower than, for example, C++, but the code is so easy that it saved by far in coding time all the time c++ in running time could win. Since Python is an interpret language the production cycle is faster.

1 Last update at the moment of writing this is 31/05

There were some discussions about which NLP tool choose; there were no discussions with Shogun, it is clearly by far the best tool available.

The other thing to be chosen was the language. Since Shogun is recommended to be used with Python (as an end user) and Python is a very good language to work with text this was also an easy pick.

So the objective now is to provide a framework to classify text from raw format with the tools described (that will be described deeper soon).

My personal objectives were:

- Learn Python [16]
- Learn basic about NLP
- Learn about feature processing
- Learn to use NLTK
- Learn how to use SVM, understand the different kernels, understand how to practically use SVM.
- Learn to use Shogun

The hardest programming part was how to use the output from NLTK as input to Shogun since both libraries are using different conventions. We will not explain much about the programming process in this part (it will be the main part in the next chapter), but as an anecdote and where we lost most of the time here was realizing that the output from NLTK (a matrix of results) has to be transposed to be used with Shogun; usually the training points are described as rows, not columns.

The work in this part was mostly about NLP not about SVM itself; it was more about how to extract the information from a text, how to represent it, how to define a feature, how to improve the results, how to make it faster. The work with SVM was more about playing with the parameters, trying different Kernels, understand how the few parameters impact on the results, realize how complexity affects the runtime, see how different feature processing provide different results. Study the classical dilemma “time vs accuracy”.

The whole process starts dealing with documents and as said it is also the main topic here; we will start with it.

3.1 Set up the framework

The first thing needed to work with documents are... documents. NLTK provides a good set of corpus to work with. We worked with three different corpuss: movie reviews (2 class), Brown corpus (5 class), chats conversations (2 class).

Most of the work will be done with the movie reviews. This corpus is provided by NLTK; NLTK provides 2000 movie reviews, 1000 are positive and 1000 are negative, the objective is to learn them and predict new ones. We chose this as the main corpus because predicting movie reviews is hard (other corpus offered a 99% accuracy out of the box, without any work, just throwing it to Shogun), while movie reviews offered around an

75% of accuracy out of the box; so it is a good set to try different techniques. Also, extend the corpus with other movie reviews grabbed from the net is easy.

Brown corpus is only used to check how SVM works with multi label sets. It is also totally provided by NLTK.

Chats conversations are used because here corpus from NLTK and corpus picked from Internet will be used. NLTK has some chats conversations about sexual predators, so we searched for “normal” chat conversations and showed how SVM can be used in a criminal scenario.

We used mainly corpus provided by NLTK because they are well-prepared to work with and classify and by saying that it is prepared to be classified we don't mean that it is easier, the opposite indeed. When working in Data Mining one of the big issues to deal with is finding a good set to classify; most of the sets we can create by ourselves are very easy. For example we tried to create a corpus about terrorism, we gathered a lot of documents about terrorists (news, FBI entries,...) and then we used other criminal related news to compare it, that news were chosen from the Reuters corpus. It was very easy to classify, without any processing we get around 99.99% accuracy.

The big and interesting real work with Data Mining is collecting useful information. In the terrorist case, a nice set would be to compare Islamic terrorism with Spanish terrorism, or terrorists from Iraq compared to terrorists from Afghanistan. In a more wide criminal scenario a good set to work would be, for example, compare sexual chat conversations with illegal sexual chat conversations. Talking about sex is legal (in most countries), illegal chatting would be when an under 18 child is part of the conversation. Would a machine be able to classify if one of the interprets is above 18 and the other under 18? That would be a good problem but we don't have access to this kind of information. The initial objective for this thesis was to classify criminal data but then we found that we were unable to use that kind of information so we had to move from sexual predators to movie reviewers.

The good point about movie reviews is that it is easy to find the documents, it is easy to label them, it is not easy for computers to classify it.

All the different tests that follows are based on the movie reviews corpus. We will try to find the best processing and the best SVM (best capacity, best kernels, best parameters). Since between NLP and SVM the number of parameters and decisions are very big we will fix the SVM part, we will only work with a Gaussian Kernel (which is probably the most used and works OK for most problems) in most of the tests, when nothing is said about the SVM we are using a Gaussian Kernel.

The other classifying method to compare the results with is Naïve Bayes because it comes out of the box with NLTK. Our objective is not to do a deep analysis about different ways to classify documents, there are already better papers about this that what we can do [9].

Nowadays there are two implementations of the SVM algorithm (we mean the most used). One of them is as said SVMlight by Joachims[10], the other is libSVM [11]. We will always use libSVM because it is GPL v3 while SVMlight is not. Shogun allows to use both and some other not so popular implementations.

About the results that will follow if nothing is said we are using a Gaussian Kernel. The results will be shown in tables because with Gaussian Kernel two parameters are required: the capacity of the SVM and the width of the Gaussian Kernel. Both parameters will be tested with different values to select the best results (rule of thumb).

For each test we will do a 4-fold cross validation. The results presented will have two values: the best and the worst result. If only one is presented it means that both values are the same. As we move forward in the tests we will find that usually some range of C[2] and some range of width provides the bests results, we will focus on them because in that way the process will be faster.

Besides the accuracy we will also show the running time. In Data Mining the running time is very important because real problems often deal with millions of entries that can take several weeks. If instead of a week we can get results in a day, although we are losing some precision, the results can probably be considered better. That is why we will always consider very important the running time.

About the result we will only consider the accuracy, that is, the number of correctly classified points between the total number of points. Some papers use the accuracy, some prefer the precision[3], some prefer the F-Measure[4].

Precision and F-Measure fit best with Information Retrieval problems because they are using the concept of "relevant", that kind of concept is hard to use in Data Mining; we can say that one class is relevant and the other class is not, but that may not our case. Are positive movie reviews relevant or irrelevant? What happens with the negative ones? In multi-class classifications, what's relevant and what's irrelevant?

We are also working with little examples, accuracy results are easy to read and understand and they not require extra calculations. There is another problem derived from the way we are mixing train and test. Since we are mixing it in a random way we can't ensure that 50% of the documents will pertain to one class and 50% to the other[5], each part in the 4-fold cross validation will have a different % of positive and negative reviews, it would be around 50%, but not exactly, this makes harder to calculate the recall.

With movie reviews we don't have a training set separated from the test set, we will use the randomizer from Python to mix both classes, we will divide that set (2000 documents) in four parts (500 documents each) in each iteration of the 4-fold cross validation one part will be considered test, and the other parts will be considered training.

Figure 1 is extracted from NLTK's book, our NLP tool. It explains the whole process

2 Capacity, the penalty we talked about in the introduction.
3 Number of relevant documents retrieved between number of documents retrieved.
4 A ratio using the precision and the recall, being the precision what said before, being the recall the number of relevant documents retrieved between the total number of relevant documents.
5 There are other ways to merge the sets that guarantee a perfect 50% mix.

about classifying anything, in our case when the “input” are “documents”. Basically we have to convert our input to something we know how to work with (real number usually) and send that information to the machine learning algorithm.

When working with SVM this process can be merged in one step, the Kernel will do something like “feature extraction” and “machine learning” in one step. In this chapter we will not do it, we will process the features outside the SVM part and it will only work as a “machine learning algorithm”. In the case of SVM the “classifier model” is a set of training points, usually a very little set. If we have thousand of training points it may be enough with a dozen (two points are enough to define a vector), that is one of the best features about SVM.

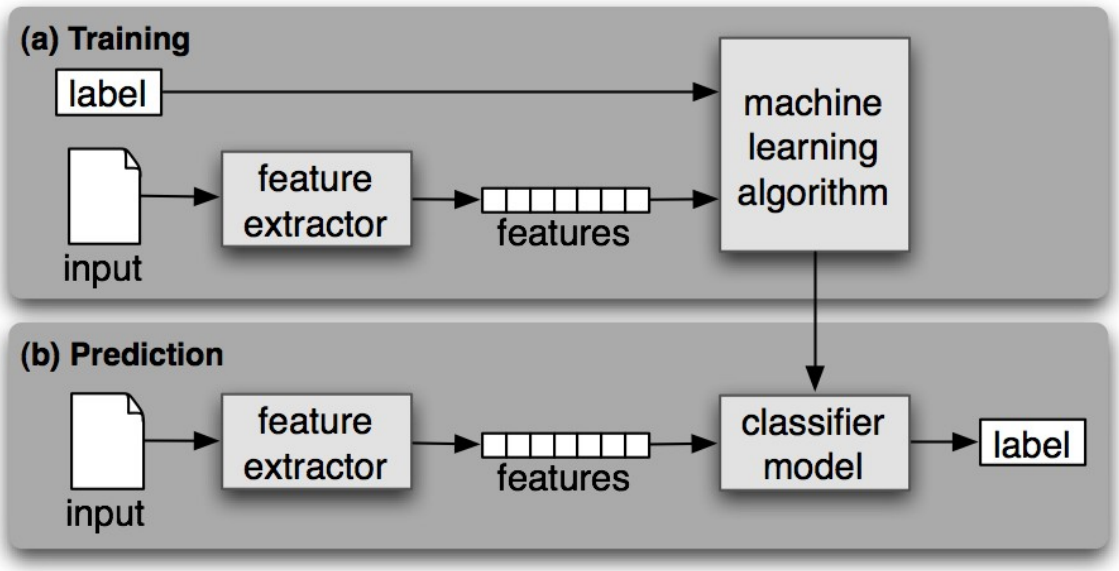


Illustration 1: Figure 6-1 from NLTK book

As required by Shogun labels will always be integer numbers; in the case of 2-class we will use -1 and +1, in the case of multi-class we will use 0,1,2,3,... in some machine learning algorithms there may be a big difference between using -1/+1 or 0/1 as labels (neural nets, for example, they compute and spread the error as the difference), in SVM it is also important, but not so important.

The “feature extractor” should be the same in the training phase and in the prediction phase otherwise some people may say that we are cheating. That's not necessary the case. Mainly we will use the same “feature extractor” but in one improvement we did we will use a different “feature extractor” depending on the phase; it will be explained later, it will be assured that we are not cheating. By cheating we mean that the training and test set are not completely separated. That the test set is using some information that should only be available to the training set.

Those all are the big details, now we are going to explain deeper each step, starting by the NLP part.

3.2 From documents to real numbers

NLP can be roughly defined as “make the computer understand a natural language”. By natural we mean “human” language. Machines have been proved to work very well with formal languages (like programming languages), it's not the case with natural language.

NLP is a big field in AI, still under big development. The number of relations that words in a natural language have is probably too big for the computers we are using nowadays; that's why in NLP we try to make problems simpler. Our objective is to classify documents, so a machine should understand the documents we will send to it and say which ones are more similar. We can make this trip easier by using “feature extraction”, formatting the documents in some way that it is easier for computers to understand it.

There are four steps that we can use to help the machine:

- Phonology: Converting from sound to text. Instead to sending the machine a .wav wave, we can send it a text, it is easier for the computer. Luckily we don't have to deal with this problem because nowadays no perfect results have been shown.
- Morphology: Once we have a text, separate it in words. It's not easy and it requires mastering regular expressions because it is not always obvious to know when a word starts and finish. Luckily there are tools that will do this for us very well. After this step instead of a text we should have a list of words. We start the process here because NLTK already provides us with list of words, otherwise using the “tokenizer” is straightforward.
- Syntax: Extracting information about the relation between words. Sometimes a word can be a noun, a verb or an adjective. With syntax we can also understand the style of writing. It is a complex field that can give us a lot of information and is not much used in DM.
- Semantics: Go from words to definitions. It can help us to say that two words are similar although they are morphology different. The problem is that when dealing with millions of words and comparisons if the computer has to search each word in the dictionary the process gets slower.

For our purposes morphology is enough.

3.3 Working with “Movie Reviews” corpus

The “Movie Review” corpus contains 2000 documents, 1000 for each class. Each document has around 800 words and around 300 different words (unique). In total there are +1.5M words and around 40K unique words. We have to think about a way to represent a document. In the next chapter we will show that there is no need to represent it, just the words are enough, but now we have to think about how to convert it to numbers, specially real numbers, that is what the Gaussian Kernel needs.

We have to extract features from it, so at first we have to define what a feature can be. In a document a feature can be the document itself, a sentence or a word. We may also think about some syntactic analysis, phonetic waves,... we will choose a word so we can also say that the most common two documents are the most words they share, we will work on that idea that has been proved to work very well [9].

If a document is defined by its words it should be enough to map one document over the other and tell how similar they are (intersection of documents), that's true, but if for each word in a document we have to find it in the other document the process gets slow. Also, in SVM we are working with vectors and inner products; imagine that we have two vectors with 500 words each one, can we define an inner product in that space as simpler as the used by Kernels?

What it's easier, we will define each document by a vector of 0's and 1's, a 0 means that the document doesn't contain a word, a 1 means that the document contains it, as explained in the introduction. Do the inner product between two vectors of 0's and 1's is easy and fast. The problem here is that there are 40K unique words in our corpus, that means that each document will be represented by a 40K vector (while a document usually have around 800 words), so that vector will have 98% of zeros, we are losing much space without information, that's not an option for a computer scientist.

Also, 40K for 2K are 80M zeros and (not many) ones. Each inner product would require 40K per 40K products, that's 160M of operations, and we have to repeat it 4M of times (2000 x 2000 to fill the kernel matrix), there are too many operations and 98% of them are just zero multiplied by zero.

3.3.1 First feature selection: Document frequency thresholding

Before starting to think about the problem, it is clear that we need a first feature selection. It is proved by [12] that “Document frequency thresholding” scales the problem to be faster to solve and also that it is not losing much information. Sometimes it offers betters results because the words removed (low frequency words) can be considered “noise”.

About our corpus:

- The 10.000 most frequent words represent the 95%.
- The 5.000 represent the 90%.
- The 2.000 represent the 83%.

We will do the first SVM execution to compare runtime and accuracy and select where to cut. As said we will use the Gaussian Kernel (GK), there are two parameters we have two control when dealing with SVM and GK: the Capacity (C) and the width (w). The capacity is a parameter required by the SVM, it was explained in the introduction, it means how soft or hard the margin is, bigger capacity means bigger penalties to miss-classified points. Usually it is better to use low capacities because it means less over-fitting and that the SVM would perform better with unknown points (it is more “flexible”). The width in the GK (and in most kernels based on distributions) means the width of the bell.

Select a good width requires an extensive statistical analysis about the data that we will not perform. We will use for width the average number of features different from zero in a training point. If for example we are working with 10.000 features (the 10.000 most frequents words) and usually only 1.000 of that features are positive in a document, we will use a width of 1.000, that makes sense because the GK is calculated as follows:

$$K(x,y)=\exp\left(\frac{-\|x-y\|^2}{width}\right)$$

On average that squared difference will be different from zero between “width” times and two “width” times. That is why on our tests we will play with the width parameter, usually it will have values between 1 and 5. The “C “ parameter will have values between 0.5 and 100. The results are as follow for 10.000 features:

The running time for 4-fold cross validation was around 88 minutes; in that time we did 48 SVM executions (4-fold C.V. and 9 tests for each one trying different values for C and W), that means 1m50s per execution. On average the width was around 310 activated features (around 9700 zero values, 97% of the feature values were useless), we are wasting a lot of space. The results represent the accuracy (number of correct classified points between total number of points), in the test phase the total number of points is 500.

w\C	**0.5**	**1**	**10**	**100**
w*1	0.472-0.54	0.618-0.706	0.632-0.756	0.632-0.756
w*2	0.682-0.744	0.77-0.87	0.774-0.876	0.774-0.876
w*5	0.772-0.858	0.808-0.888	0.834-0.892	0.834-0.892

The set of parameters chosen would probably be “w*5” and “C=10” although for a real world scenario the best would be “w*2” and “C=1” because it also offers good results and the bell's width is in the range expected and the margin is not hard. We can also see that there are big differences between the results we get in different executions because we probably have a lot of noise.

These are the results for 5000 features (the most 5000 frequent words):

The running time for 4-fold C.V. was around 44 minutes, we did the same number of executions as before, that is 55 seconds for executions. Width was on average 285 (around 4700 zero values, 86% of the feature values were useless). We are still wasting space but we are saving a lot compared to the execution before.

w\C	**0.5**	**1**	**10**	**100**
w*1	0.476-0.58	0.708-0.742	0.738-0.76	0.738-0.76
w*2	0.746-0.766	0.812-0.858	0.826-0.868	0.826-0.868
w*5	0.82-0.854	0.85-0.876	0.848-0.87	0.848-0.87

Comparing both outputs, in 10000 the best results are better but they also have a bigger standard deviation, in 5000 the best results are worse but they are still very good and the standard deviation is much lower, probably because we removed some noise; the average results we can expect are very similar. Statistically the 5000's can be considered better, it is also saving a lot of space and it takes half time.

These are the results for choosing 2000 features:

The running time was around 17 minutes; the time per execution was 21 seconds. Width was on average 237 (around 1570 zero values, 78% of the feature values were useless). We are saving a lot of space and being more productive.

w\C	**0.5**	**1**	**10**	**100**
w*1	0.464-0.482	0.718-0.816	0.742-0.826	0.742-0.826
w*2	0.754-0.83	0.826-0.866	0.84-0.868	0.84-0.868
w*5	0.814-0.862	0.822-0.872	0.838-0.86	0.838-0.86

The best results are similar and the standard deviation is also similar, moreover we are getting the best results for a width around "*2" which is what we expected. The process is much faster, we are saving a lot of space; we consider this the best solution.

So, as our first "feature selection" we will only use the 2000 more frequent words. They represent the 83% of the information.

We will now try the same scenario for Naïve Bayes; the accuracy we got was 0.79 – 0.81, worse than SVM. The run time for one execution was 62s (compared to 21 seconds). One interesting feature about Naïve Bayes is that it can tell us the best feature, the list is as follows:

contains(word)	**Relevance ratio**
contains(outstanding)	pos : neg = 13.6 : 1.0
contains(seagal)	neg : pos = 11.7 : 1.0
contains(lucas)	pos : neg = 7.8 : 1.0
contains(mulan)	pos : neg = 7.7 : 1.0
contains(wasted)	neg : pos = 7.1 : 1.0
contains(jedi)	pos : neg = 6.2 : 1.0
contains(waste)	neg : pos = 6.0 : 1.0
contains(awful)	neg : pos = 6.0 : 1.0
contains(ridiculous)	neg : pos = 5.6 : 1.0
contains(poorly)	neg : pos = 5.6 : 1.0

Table 1: Results provided by NLTK's Naive Bayes about relevant words

Steven Seagal wouldn't probably be happy.

3.3.2 Removing non-aplhanumeric and stopwords

If we check which are the 20 most frequent features, we get the following results:

Word or symbol	**Number of appearances**
,	58314
the	57311
.	49440
a	28442
and	26458
of	25592
to	23757
'	22920
is	18854
in	16367
s	13907
"	13283
it	12119
that	11966
-	11617
)	8691
(	8618
as	8478
with	8074
for	7466

Table 2: Top 20 words or symbols provided by NLTK tokenizer about Movie Reviews corpus

First we can notice that non-alpha numeric features are very common and it is not necessary any analysis to know that a comma would be as common in positives reviews as in negative reviews. The other thing we can see is that the other features are very common words, like “the”, “and”, “in”,... it is also not necessary any analysis to know that this words will appear in probably the same quantity in both classes. These words are usually called “stop words” and are usually removed because they are not giving any information, also we will remove commas and so on; we will only work with alpha-numeric words. Removing stop-words and non alpha-numeric words is probably the first step when working with texts in any field.

Now in total we have 710K words and on average each document has around 350 words (1.5M and 800 before). We can be sure that we have not lost information, and now our corpus is smaller; that means that we can work faster. Let's check the 20 most frequent words now:

Word or symbol	**Number of appearances**
film	7415
one	4492
movie	4258
like	2724
even	1860
good	1830
time	1822
story	1714
would	1605
also	1595
character	1531
well	1526
much	1520
two	1482
characters	1434
first	1420
get	1402
see	1348
life	1337
way	1296

Table 3: Top 20 words after removing Stopwords and non-alphanumeric

This makes more sense. We will do another execution of our SVM, with 2000 features. As before each feature will be 0 or 1 if that document contains (or not) the word:

The average width now is 166 (before it was 237), it means that for each document 70 words were stopwords or non-alphanumeric, this also means that our feature vectors will be again full of zeros. The run time is still 17m because the number of feature is the same.

w\c	**0.5**	**1**	**10**	**100**

1*w	0.466-0.492	0.662-0.756	0.68-0.766	0.68-0.766
2*w	0.71-0.778	0.802-0.842	0.816-0.854	0.816-0.854
5*w	0.814-0.854	0.84-0.848	0.844-0.862	0.844-0.862

The best results are worse but the worst results are better; the average results are probably the same and the standard deviation is smaller (near zero in some cases) because we have removed most of the noise. This solution would be considered better.

In all these tests we can see that for C=10 and C=100 the solution is the same. That is because the SVM is already overfitted at 10 but not always it will be like this. We can also check that for “1*w” the results are bad and that's because as we said before “1*w” can be considered the lower bound of the solution. From now we will start with “w=w*1.5”. We will keep C=10 and C=100.

We will check the Naïve Bayer results which are 0.78-0.82 and the run time per execution is 55s, 7s seconds faster than before and better accuracy.

3.3.3 Normalizing words

The next step when dealing with a text is normalizing it. If for example one document has the words “film” and “films” it would be considered two different words while it should not because it refers to the same concept. If we have to classify three documents, one about “a tree”, the other about “trees” and the last about “flower”, without normalizing it those three document would be considered different while they are obviously not.

There are two techniques to normalize a text, stemmers and lemmatization.

An stemmer would try for each word to only save the lexeme. In the case before the lexemes would be “tree”, “tree” and “flower” so it would be easy for the computer to classify it. It is not a defined process, stemmers use regular expressions to extract lexemes; it means that they can create lexemes that don't exist or that for two words with the same lexeme they can create two different ones. In the case of “tree” a “trees” and bad stemmer would create “tre” and “tree”.

The lemmatization is a process similar to the described before, but it checks that the output exists in a dictionary otherwise it discards the word so the output is always readable. When we saw the popular top features we could read that “Seagal” and “Mulan” were good features however they would be completely useless in a real world scenario; that words are removed because they don't exist in any dictionary. Stemmers are faster though we will use lemmatization; all this process is provided by NLTK, as usual.

After using it to our corpus, we have 645K words compared to 710K. We will do the SVM executions with 2000 features as usual. We don't expect to have better results than before because our current feature extractor (be or not) is not exploiting this idea. Run time is the same as before.

w\c	**0.5**	**1**	**10**	**100**
1.5	0.622-0.64	0.794-0.8	0.796-0.82	0.796-0.82
2	0.732-0.736	0.826-0.828	0.828-0.84	0.828-0.84
5	0.836-0.848	0.844-0.854	0.85-0.858	0.85-0.858

The results are very similar, but now the standard deviation is even smaller and that would be considered better because with lemmatization we removed even more noise. With other feature extractors lemmatization will be better used.

3.3.4 Feature selection: information gain

Now we arrive to the last stop in our path about pre-processing the documents. The objective in all these steps was to make the corpus as small as possible without losing information and maybe gaining accuracy by removing noise. The next step is the most drastic one because we will remove a lot of features. First we need to look again to the 20th most frequent words, they are:

“'film', 'movie', 'one', 'like', 'character', 'make', 'get', 'see', 'scene', 'even', 'good', 'time', 'story', 'go', 'much', 'play', 'well', 'also', 'take' and 'two'”.

If we think about movie reviews we can guess that some of this words are not providing useful information. For example “film”, can anyone with the word film decide if a movie is good or not? This step is called “information gain”; we have to choose the set of features that provides more information.

What follows is again the top 20 list with a ratio that means: number of times that the words appears in positive reviews between total number of times (this value about negative reviews will be 1-ratio). The more this ratio is around zero the less information this word is providing:

word	**P{+|word}**
film	0.55
movie	0.45
one	0.52
like	0.49
character	0.53
make	0.49
get	0.47
see	0.57
scene	0.50
even	0.45
good	0.52
time	0.50
story	0.57
go	0.51
much	0.50
play	0.52
well	0.59
also	0.60
take	0.55
two	0.52

Table 4: Words and its probabilites to be in a positive review

It is hard to analyze what is inside a movie critic's brain but it seems that when the movie is good they use the word “film” otherwise they prefer “movie”. We can also check some other words they use for good movies like “also”, “well”, “story”, ”see” and some words that are not providing information and should be removed like “much”, “time”, “scene”,...

In the multi class corpus we will do a more powerful information gain feature selector using chi^2 statistic. In this 2-class example we will use the ratio used before: we will remove any word that has a value between 0.55 and 0.45. Now the total number of words is 396K (before around 645K) and each document has around 198 words. Let's see how this performs with SVM:

The average width is around 106 words. This means that again we have feature vector full of 0's, around 99% of the space are 0's. The run time is the same because we are still using 2000 features.

w\C	**0.5**	**1**	**10**	**100**

1.5	0.656-0.688	0.756-0.806	0.772-0.814	0.772-0.814
2	0.726-0.77	0.798-0.826	0.81-0.82	0.81-0.82
5	0.82-0.838	0.828-0.83	0.82-0.842	0.82-0.842

The results are good although slightly worse. While before the best accuracy was around 0.85 now it is around 0.83. The good side about this feature selection is that it allows us to use less features, 2000 are not anymore necessary because 99% of them are zero. We will perform the same test using only the best 1000 features:

The average width now is around 90 words (and we are using half features, that is a good sign, we are using around 10% compared to 1% before). The run time now is 12m compared to 17m before. That means 15s per execution (21s before).

w\C	**0.5**	**1**	**10**	**100**
1.5	0.656-0.68	0.774-0.79	0.798-0.802	0.798-0.802
2	0.74-0.754	0.824-0.828	0.826-0.846	0.826-0.846
5	0.818-0.828	0.824-0.836	0.824-0.83	0.824-0.83

As expected the results are slightly better with lower standard deviation and with better results were "w" is twice its value. Before the best accuracy was around 0.85 and now it is around 0.836 though the execution is faster (15s vs 21s).

In our first test the corpus had more than 1.7M words and each document had around 800 words; we were using 10000 features and the run time was around 88 minutes. The best results were around 0.85.

Now our corpus has 396K words and each document has around 198 words. It took 12 minutes and the best results are around 0.836 while we are saving a lot of space and a lot of time losing 0.014% accuracy. We can call this a good result.

The time vs accuracy dilemma depends on the kind of problem. The problem defines how important the accuracy is. If we are dealing with important data were the accuracy is the most important thing (like medical tests) we can't do this kind of feature selection, otherwise if we are dealing in an scenario were more errors (we are talking about 0.014%) are not much important, we can say that this solution is better because we are saving a lot of time and space.

If we check now the top 20 features we get the following results:
"'film', 'character', 'see', 'story', 'man', 'also', 'life', 'take', 'first', 'well', 'way', 'plot', 'people', 'love', 'look', 'star', 'best', 'show', 'become', 'bad'".

Some of this words are clearly good features to decide if a movie is good or bad. Film, for example, is good at the training set but, would it be a good feature in real word scenario? Probably not.

3.3.5 How to represent a document

Until now there have been two constant constraints. First as we said we will only work with Gaussian Kernel[6]; we will later discover if some other kernel would provide better results. The other constraint was the way we represent a document. We only said if a document contains a word or not. That is probably the most basic way to do it. However it can easily be seen that it is not the best way and that with a little work we can get better results. The first update we can do is instead of saving if a word appears or not to count the number of times it appears. This will also use better the lemmatization process. The number of features will be the same so the process will not be slower and the results are expected to be better.

Well, they should be better when working with big documents, but we are working with movie reviews. We don't expect to get a big improvement when doing a multi intersection. On our case, on average only 20 features have a value bigger than 1 (20 over 1000) while around 80 have a value of 1, so it will not make a big difference.

If we focus on the Gaussian Kernel:

$$K(x,y)=\exp\left(\frac{-\|x-y\|^2}{width}\right)$$

We notice that it is computing the difference between the training points so GK does not care if a word appears 7 times on one document and 7 times in the other: it would be a 0, the same result that would have if it appears once in both documents. It is clearly not using the feature vector as we would like to. It is calculating the distance while what we need is “how similar they are”. Another famous kernel is the Chi^2, it is also not much useful because it is still computing a distance so we will try the most basic kernel: the linear kernel:

$$K(x,y)=x\cdot y$$

We didn't get any improvement. We also tried the Sigmoidal Kernel without any improvement too; maybe we are in the wrong way. The next step we tried was using the term frequency instead of the number of appearances. That would fit better in kernels that focuses on the distance. The results are quite similar so we will not write them again.

When working with documents in Information Retrieval or Data Mining the most common way to represent a document is the “tf-idf” weight. It is an update about term frequency (tf). The idea is to use another value (idf) to weight the term frequency. Idf means “inverse document frequency”. While tf is a value about a document, idf is a value about the whole corpus. Idf says how popular in a corpus a word is; it tries to mark the words as relevant and non-relevant. The most popular a word is, the less relevant it is.

It makes sense in information retrieval. If someone searches in Google “football Odense” it can be seen that “Odense” (probably with a much higher IDF than “football”) is a better

6 Considered the swiss army knife

word to decide which documents are relevant and which are not. It is better to search for all the documents about “Odense” and later among them search for “football” than doing it in the opposing way.

Will it be useful in DM? We could see in our TOP 20 lists of words that some of them are very popular and spread among all the classes. The idea is to find the words that are very popular inside a document but rare in the whole corpus; that words would define that document.

The most popular a word is, the lower IDF value it has; we do the dot product between tf and IDF. The formal definitions are as follows:

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}}$$

For a term “i” in a document “j”.

$$idf_i = \log \frac{|D|}{\{d : t_i \in d\}}$$

The total number of documents between the number of documents that contains the word “t”.

Finally:

$$(tf - idf)_{i,j} = tf_{i,j} \times idf_i$$

The results with “tf-idf” are as follows. We will use again 2000 features, we will only remove stop words and non-alphanumeric and use a lemmatizator:

w\c	**0.5**	**1**	**10**	**100**
w*1.5	0.48-0.5	0.48-0.5	0.56-0.68	0.81-0.82
w*2	0.48-0.5	0.48-0.5	0.49-0.52	0.8-0.83
w*5	0.48-0.5	0.48-0.5	0.48-0.5	0.79-0.81

The results with tf-idf are worse than just checking if a document has or not a word. That is because we are dealing with a little corpus with not many documents and each document with not much words. As we said before a document has around 300 words and only 20 of them appear twice or more times and we are using 2000 features, 20 over 2000 is nearly nothing. We are not using the true power of “tf-idf”.

3.3.6 Collecting new data

A Data Mining problem can arise from two different directions: we may have some data we want to classify or we may have some data to work with and we may think about using Data Mining. “Data” can be anything from documents to robot's movements; also Data Mining has two different fields: “classification” and “regression”. We are classifying now although SVM and most DM methods can be used in regression too.

The other different direction is when something theoretically is wanted to be proved. We will do that in the next chapter while now we want to show a framework to classify documents. We don't have the data, we have to search for it and that is probably one of the most difficult task. It is not only about finding data, it is about finding good data.

Luckily in our problem “movie reviews” find new data is an easy task. We have been working only with the corpus provided from NLTK and we want to test our system now with a real world scenario. To collect the data we went to the website “www.rottentomatoes.com”, in that website reviews are classified as positive and negative. We chose 50 positive reviews and 50 negative ones without reading it. We only focused on reviews with more than 400 words; from the same movie we chose positive and negative reviews.

We kept the process already done: dividing the NLTK corpus in a 4-fold cross validation and for each test we are checking it against our new corpus too, so our new corpus is not part of the training set.

Each new review is a “txt” file containing “html” code copied from the website. We use NLTK to “clean” the “html” code and then we use NLTK's tokenizer to convert that clean document to a list of words. With that two easy steps we convert a document from the real world to something useful for our system.

We will only show the results for the new set, the results for the NLTK are as showed before for the same scenario. We are choosing 2000 features, removing stopwords and non-alphanumeric words. We are using “be or not” to represent the documents:

w\C	**0.5**	**1**	**10**	**100**
1.5*w	0.82-0.83	0.78-0.82	0.77-0.83	0.77-0.83
2*w	0.82-0.83	0.8-0.82	0.77-0.83	0.77-0.82
5*w	0.83-0.85	0.82	0.82-0.83	0.82-0.83

These results are really interesting because they show that when dealing with real-world data a soft margin performs better that a hard margin (it was the opposite with the training set because the system was overfitted).

Now we are going the try the same set under the same constraints though now using “tf-idf” weight to represent documents:

w\C	**0.5**	**1**	**10**	**100**
1.5*w	0.5	0.5	0.53-0.77	0.87-0.92
2*w	0.5	0.5	0.5	0.88

5*w	0.5	0.5	0.5	0.87-0.88

These results are again interesting. The 0.5 means no training although with a hard margin we are getting the best performance we have ever had[7].

We are getting no training with soft margins because “tf-idf” needs more documents and more words to represent them correctly. If we look through the kernel matrix the values are “0.999x” were x is where the entries are different. That means that the GK has to classify everything with very little information, this is why with soft-margins “it gets crazy”.

The results with other common kernels like “Chi^2 Kernel”, “Sigmoidal Kernel” or “Poly Kernel” are very similar so we will not reproduce them because they are not giving any new information.

This was all the work we did with the “movie review” corpus. The objective was learning to use NLTK and Shogun to provide a way to classify documents. We worked only with SVM but Shogun has other methods, these methods were not explored because the objective was not to make a comparative study.

On average all the script files we used to make tests had around 100 lines.

3.4 Working with the “Brown corpus”

Our objective now is double: Test SVM with a multi class set and improve “tf-idf”.

We will not reproduce all the process as we did before because it would be redundant however all those steps were done again for the new corpus. We will use the same constraints as before if we don't say the opposite.

The Brown corpus is not prepared for DM: it is just a collection of documents about different topics. We picked the five most populated topics (the most data the better results). That five topics are: “learned”, “belles lettres”, “lore”, “news” and “hobbies”.

Then we removed, as usual, the stop words, the non-alphanumeric words and we used a “lemmatizator”. We merged all the documents related to a topic and then we cut each 185 words, in this way we get 200 documents about each topic with 185 words each one, 185 words is not much but after removing all the “noise” it may be enough.

We merged all the documents in a big set, randomize it and made 4-fold cross validations. What follows are the results for the “be or not” document representation, we start by choosing a feature vector of 5000 features (as before, the 5000 most frequent words, they represent about 90%):

w\C	**0.5**	**1**	**10**	**100**
1.5*w	0.224-0.292	0.688-0.784	0.696-0.796	0.696-0.796

7 The results for NLTK training/test set were as always, as we said “tf-idf” was not improving “be or not”

2*w	0.352-0.392	0.72-0.808	0.756-0.82	0.756-0.82
5*w	0.692-0.748	0.756-0.82	0.804-0.852	0.804-0.852

It took around 32 minutes. The results vary more than usual because it is harder to classify when you have more classes (in our case 5). Anyway the results can be considered good. Also, only around 120 features were activated among the 5000. That means that we are wasting around 97% of space, probably too much.

We will try now the same but with 3000 features instead of 5000. That top 3000 words represent the 82%. It took around 17 minutes, around half time than the test before. On average 110 features were activated. That means around 96% of wasted space, not much improvement:

w\C	**0.5**	**1**	**10**	**100**
1.5*w	0.224-0.332	0.692-0.724	0.704-0.728	0.704-0.728
2*w	0.416-0.44	0.732-0.764	0.74-0.76	0.74-0.76
5*w	0.684-0.744	0.752-0.772	0.776-0.784	0.776-0.784

The results are worse. This feature reduction was not good because since we are dealing now with 5 classes the number of features we need is bigger. We will try now 5000 features using the “tf-idf” representation for documents. In the previous corpus “tf-idf” proved to be better for real-word corpus (that's what is really important) but it didn't offer better results for the train/test set:

w\C	**0.5**	**1**	**10**	**100**
1.5*w	0.176-0.18	0.176-0.18	0.2-0.3	0.92-0.932
2*w	0.176-0.18	0.176-0.18	0.184-0.344	0.904-0.928
5*w	0.176-0.18	0.176-0.18	0.176-0.18	0.86-0.884

It took 28 minutes. We can see now the real power of “tf-idf”: the results are much better than “be or not”. We also tried to use 3000 features although the results were worse so we may need another kind of technique to improve our results.

3.4.1 Improving tf-idf

Although the results with tf-idf are good enough we think that the formulation itself has a handicap [13]. With “tf-idf” we know if a word appears of not in a document (as the binary representation), we also know how many times it appears and we also know if that word is relevant or not in the corpus, all in one number. There is, however, something we think that can be improved about “tf-idf” because it was created for IR not for DM.

IDF tries to say if a word is relevant or not. It is good in the sense that in the “movie review” context it will mark as non-relevant words like “film”,”movie”,... but it is not good in the sense that it will mark as non-relevant words like “good” or “bad” because those words are very common in the corpus.

If “good” appears in, let's say 700 documents, it will have a very low IDF, it will be marked as low relevant feature but we know that it isn't. We know that in our context if a document contains the word “good” we can say that that document is probably a positive review. IDF makes relevant a lot of low frequent words and forgets a lot of high frequent words that in our context should be considered very relevant.

We will create a new parameter called “idf class” that represents how important is a word for a class. It is defined as number of appearances of a word (i) in a class (j, defined by the label of the document containing that word) between the total number of appearances of that word (i):

$$IDF\,class_{i,j} = \frac{|t_{i,j}|}{|t_i|} = P\{class=i \mid word=j\}$$

The more this number is near 1 the more this number is important for that class. If we are dealing with two classes a IDF class over 0.5 will mean some kind of relevancy, with three classes more than 0.33. We have to determine some kind of threshold to define what makes it relevant and what not.

Tf-idf is rewritten as:

$$(tf-idf)_{i,j} = \frac{tf_{i,j} \times idf_i}{(1-IDFclass)}$$

IDFclass can have a value of 1 that would make a division by zero and that also would make the feature vector too sharp (big differences between values). That is why it may be better to choose a bigger value than “1” like “1.5”. In that case at much we will double the “tf-idf” value and we will also get an smoother feature vector. That value would depend on the number of classes.

We can represent the documents with this new version of “tf-idf” but there is a problem here: how we convert the documents in the test phase? In the test phase we receive documents without a label, how can we calculate the IDFclass value if for a given word in the new document we can't say which class it belongs to?

One solution is to use a normal “tf-idf” representation for the test phase, but that would create different feature vectors and the results will not improve anything.

Another better solution is to use an statistic to predict given a word in the test phase and the train set which class it can belong to. We will use the chi^2 statistic:

$$\chi^2(t,c) = \frac{N \times (AD-CB)^2}{(A+C) \times (B+D) \times (A+B) \times (C+D)}$$

This value for a term (t) and a class (c) where:

- A is the number of times t and c co-occur.
- B is the number of times t occurs without c.
- C is the number of times c occurs without t.
- D is the number of times neither c nor t occurs.
- N is the total number of documents.

So for each of the 5000 features we calculate to which class they "belong", by "belong" we mean that it has a higher chi^2 value.

Later when we receive each word of the documents we will use that chi^2 value to decide the best class and calculate the IDFclass. As we said in the beginning of this chapter we are using a different feature selector depending on the phase.

We tried this new document representation and the results were slightly better (against the 5000 features tf-idf), but not enough and not so constant to say that they are really better. That is why the results will not be reproduced on a table. Getting the same results is good in the sense that it means that our method is working, however it is bad in the sense that we expected to get better results. We believe that this new document representation would produce better results although we didn't get them, maybe because we are working with a little set.

Now, with the chi^2 statistic we can perform another feature selection based on information gain. The chi^2 statistic tells us how relevant a word is to each class, so we can remove from our features that words that are not much relevant for any class. We will perform this feature selection after the top 5000 words are selected. Between that 5000 words we will remove those that are marked as non-relevant by chi^2. Different thresholds can be tested, for example if we remove that words that had less than "2" as best chi^2 (between all the results for all 5 classes, we saved the best) only 500 features of 5000 will be removed, if we set that threshold to 3, 1000 features will be removed (so we will work with 4000 features).

We will only focus on the results for C=(10,100) because we know that for the other values the results will be bad. These are the results we get after removing stopwords, non-alphanumeric words, selecting the top 5000 words and removing from them that with a chi^2 best score below 3. We use the tf-IDFclass (our version) as document representation:

w\C	**0.5**	**1**	**10**	**100**
1.5*w	🟥	🟥	0.692-0.704	0.936-0.944
2*w	🟥	🟥	0.356-0.504	0.94-0.948
5*w	🟥	🟥	0.172-0.188	0.92-0.936

We can see that the results are slightly better and the process much faster (4000 features vs 5000 features). We are removing noise and we are using the chi^2 statistic twice, that's maybe creating some synergy between the results. Anyway these results are not enough good and further analysis should be required. Sadly in this Master thesis we can't stay here longer and we have to move forward to the next topic because we don't have extra time. Improving the tf-idf is a “side quest”. What is really important about improving would be shown in the next chapter.

3.5 Chats

After working with two corpus, we have the experience to repeat this process faster. We know the best way to represent a document, how to extract features, how to select them. We will not repeat all of this again.

The chat corpus will be 2-classed: one will have chats where sexual predators are involved and the other will have normal chats. The chats are divided in documents each one containing 70 words; in this way we obtain 400 documents, 200 for each class.

Stopwords and non-alphanumeric words are removed, we lemmatize it and we also remove common words for chats, like “room”, “leave”, “enters”,...

If we chose of that 400 documents 100 to be the test and 300 hundred to be train we get an accuracy of 99.99%, this means that it is missing only 1 from 100.

Our objective will be different now. We want to have a training set much smaller than the test set because that is what is happening in real world. We would have a SVM receiving chat logs and it has to say if that chat logs are about predators or not so the test set will be much bigger. From that 400 documents we chose 40 to be the train, 360 to be the test. We get a result around 95%, quite good.

Of course our set is not the best one: we are comparing sexual predators with normal conversations, they are a completely separated worlds. How will SVM work comparing sexual predators with legal chat logs? That would really be a good answer.

3.6 More about document representation

We have talked about four ways to represent a document: “be or not”, “term appearances”, “term frequency” and “tf-idf” (with our own version). “Tf-idf” proved to be the best.

All this four ways have something in common: they are only focusing on the words. It doesn't matter if the documents make no sense (although this is good for cryptography). Also, they are not working with the relation between words inside a document and they also don't care that some “entities” can have different names. The results are good, but it can be clearly seen that our tools are not the best and actually “tf-idf” was not created for DM. It was created for IR.

A quick fix would be to use a windows of 2 words instead of one. In that way we will also save the information about which words are usually together. The bad part is that the number of collisions is much lower although a mixed method can be tried.

Another better way to fix it is to use a sentence as window. Which words usually occur in the same sentence? The number of collisions is much lower and also the computation time is bigger. But in that way we could get a better feeling about the relation between words.

Finally, we compute the documents doing “intersections”, roughly searching for how many words they have in common, but we know that the words can be related without being the same (having the same lexeme). For example if we have three documents: one talking about trees, the other talking about flowers and the other talking about cars; our method would produce no similarity while we know that the documents about flowers and trees are more similar.

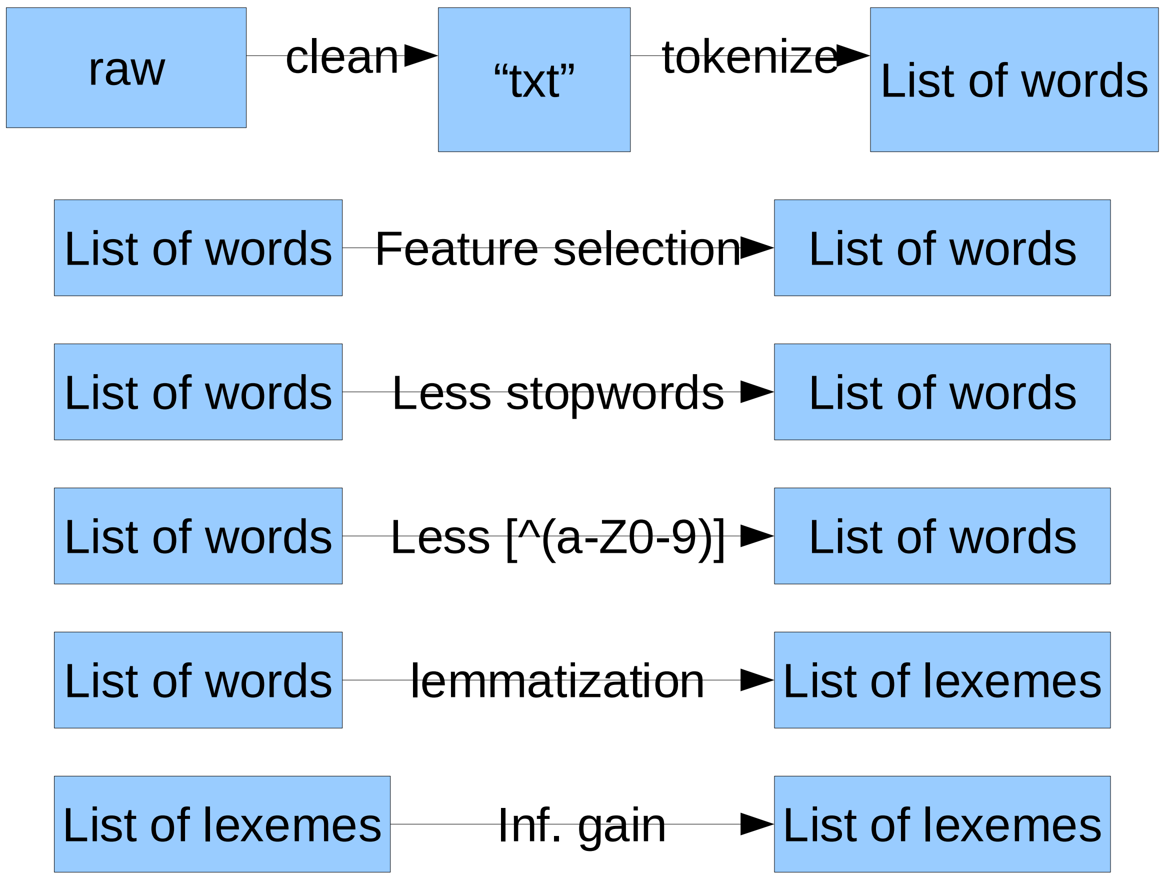
There is a web of words called “Wordnet” that has some functionalities about words. The ones that are important here is that Wordnet has different ways to say how similar two words are based on their hierarchy of concepts; usually how much of the hierarchy tree they share.

We tried to implement this, however the implementation of Wordnet is quite slow nowadays. If just doing the intersection took some minutes in most cases, for each pair of words searching how much they share in the hierarchy tree was too much. We let the program work for one day without getting any results, it was still calculating it. We think that with a faster implementation and caching techniques this method can be reproduced, but as it happened before we don't have time to stop here.

With this we finish now this chapter. In the next chapter we will move our point of view from end-user to developers. While this chapter was full with tables about results next chapter will be more about design decisions.

3.7 Summary

To end this chapter we will review the NLP process again:



We start with some documents in some format (HTML, pdf, doc,...). We use some tool (sometimes called cleaner) to convert it to a ".txt" file (plain text document where each byte contains an ASCII code).

We tokenize that "plain text" document. This means that we separate that "long string" in words and store them in a list or some structure we can work with in our programming language.

Now starts the NLP part (although the NLP we are doing here is very basic, actually this process may not be called NLP). We select to work only with the top XXX most frequent words. We remove some noise (stopwords and non-alphanumeric words), at this point we can say that we have not lost any information and that the results should be the same before this process.

In our next step we try to get more collisions by only saving the lexemes (or lemmas, lemma is the entry in the dictionary, a convention to represent several words with the same lexeme). In some language like Spanish where all the words have different morphemes depending on the context this process is highly important. Most of the time with this process we have not lost information, we may gain it, but sometimes we may lose because two different words representing two different entities with different lexeme (or lemma) can get the same lexeme if our "lexeme extractor" is not the best, and it is impossible to make a perfect lexeme extractor because that would mean to add some rules that only apply to a word.

For example in Spanish in some cases both English and Spanish words are accepted for the same entity and we don't mean jargon words, we mean truly accepted words by our normalizer institution. Examples are “parking” and “aparcamiento”, “computadora” and “ordenador”, “whisky” and “guiski”, that may do this process as harder as impossible.

Finally we have the most risky process: feature selection about their “information gain” or “how important is a word for our problem”. We try to find the words that are better to classify documents. In our case (movie reviews) it was clear that “film” is a bad word, however in other scenarios “film” can be a very good word.

As we said before we can extract more information from a document like relations between words, definitions, syntax,... that process are a lot harder than what we did and we still got good results.

4. Making texts similar

Until now we have worked with SVM as a user; SVM were a black box that magically would classify everything for us. In the introduction we explained in a very basic way how SVM work. The most important concept about SVM related to our work is the Kernel. A Kernel has two objectives:

- Transform the data from an input space to a feature space (where the next objective can be performed).
- Compute the inner product between training points.

The hard part creating a Kernel is the transformation. It is where we have to use our imagination (or our knowledge). We have some data in an input space. In that input space we can probably solve the problem, but we don't know how to make a machine to solve it so we project it to a space solvable by a machine.

Theory about SVM says: The kernel projects the data from an input space where the problem can't be solved to a feature space when it can be solved. They usually call this “to simplify the problem”. By simplify they mean “to use a feature space when the problem can be solved by a linear function”. Although SVM are part of DM, and DM is part of AI, it is really more the kind of work a math person would do. That is why we have this kind of formulations.

They consider that an input space must be projected to a new feature space if they can't (or don't know how to) apply a linear function in the input space. What usually happens is that we have a problem that is human solvable, but theory says that we need a linear function so we don't care if a human can solve it or not. We use our theory that has been proved to be valid, project it and solve it

Also we need an space that can perform inner products because that is the way SVM says how “similar” two training points are. Maybe in the input space a human can say how similar two training points are and it can be something more complex and better than an inner product.

Most SVM problems can be solved by a human without doing the two steps required by SVM. Of course machines are a lot faster so we need them, but maybe we can tell the machine how a human would solve the problem and let the machine do the hard work for us. This kind of formulation is more similar to AI (the AI computer people like). Give the machine a human behavior, let it do the hard work for you.

Would that work? That's what this thesis is about.

We are not defining a new DM method. We are only changing a bit the formulation about SVM. Our objective is to use a SVM tool (Shogun) and apply our method. This has to be done with the less number of changes possible, the less we change the most probably it will work (we use this idea in the SVM theory and in the SVM implementation). We will start talking about what will be changed in SVM theory:

In the introduction we explained how to go from the problem basic formulation:

$$\begin{aligned} &\textit{minimise}_{w,b} && \langle w \cdot w \rangle \\ &\textit{subject}\text{ to} && y_i(\langle w \cdot x_i \rangle + b) \geq 1, \\ &&& i=1,\ldots,l \end{aligned}$$

To something easier to solve:

$$\begin{aligned} &\textit{maximise} && W(\alpha) = \sum_{i=1}^{l} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{l} y_i y_j \alpha_i \alpha_j \langle x_i \cdot x_j \rangle \\ &\textit{subject}\text{ to} && \sum_{i=1}^{l} y_i \alpha_i = 0 \\ &&& \alpha_i \geq 0, \;\; i=1,\ldots,l \end{aligned}$$

This is easier to solve because the solution is a linear combination of the training points. It was explained with “more” detail in the introduction and it is very well explained in [4]. The problem with that formulation is that it only solves the problem if it is linear separable (with a linear function). So, what happens if the problem is not linear separable? (or we don't know if it is). Since we have this formulation that works, theory says that we must transform our problem to something linear separable and that means projecting the data. The kernel is doing so and it requires to rewrite the formulation:

$$\begin{aligned} &\textit{maximise} && W(\alpha) = \sum_{i=1}^{l} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{l} y_i y_j \alpha_i \alpha_j K(x_i, x_j) \\ &\textit{subject}\text{ to} && \sum_{i=1}^{l} y_i \alpha_i = 0 \\ &&& \alpha_i \geq 0, \;\; i=1,\ldots,l \end{aligned}$$

But there may be another solution because there are two things we can play with: the space or the separable function. SVM theory chose to change the space while we choose to change the separable function:

$$\begin{aligned} &\textit{maximise} && W(\alpha) = \sum_{i=1}^{l} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{l} y_i y_j \alpha_i \alpha_j S(x_i, x_j) \\ &\textit{subject}\text{ to} && \sum_{i=1}^{l} y_i \alpha_i = 0 \\ &&& \alpha_i \geq 0, \;\; i=1,\ldots,l \end{aligned}$$

S(x,y) is a function that is computing the similarity value between the training points in the input space. S(x,y) says how a human would say how similar that two points are. SVM would acquire that knowledge and solve the problem. S(x,y) can be anything if it returns a value about the similarity. That value can be anything like a new image or a text description if we want to compute the similarity between two pictures. That is what a human would do.

Since that idea is beyond this thesis we will make S(x,y) return the same that a kernel would return: a real value. This makes the problem simpler although we are losing some information[8]. That is how things are working and rewriting it would require probably years of works and new theories.

4.1 Can we use any similarity with SVM?

Not all, but most of them are good.

SVM mathematical formulation has two big constraints related one with the other: We are working in a space with inner product (the feature space) and we have to satisfy Mercer's Conditions[9]. So we can say that the similarity function must be a “masked” inner product and that constraints it quite a bit. It has to be an inner product in the usual formulation because it is using an Euclidean space (the feature space; an inner product space).

What we are calling “similarity” is commonly called “proximity index” or “proximity function”. There are two kinds of proximity indexes: similarities $(s_{i,j})$ or dissimilarities $(\delta_{i,j})$.

As said in [14] a proximity index $(p_{i,j})$ has to fulfill the following properties:

1. Non-negativity. The $(p_{i,j})$ cannot be negative.

$$s_{i,j} \geqslant 0$$
$$\delta_{i,j} \geqslant 0 \, \forall \vec{x}_i, \vec{x}_j \in X$$

2. Symmetry. The $(p_{i,j})$ do not depend on the order of i,j.

$$s_{i,j} = s_{j,i}$$
$$\delta_{i,j} = \delta_{j,i} \forall \vec{x}_i, \vec{x}_j \in X$$

3. Boundedness. There is a maximum similarity and a minimum dissimilarity.

$$s_{i,j} \leqslant s_{max}$$
$$\delta_{i,j} \geqslant \delta_{min} \forall \vec{x}_i, \vec{x}_j \in X$$

4. Minimality. The extreme values are attained for equal objects, and only for them.

$$s_{i,j} = s_{max} \Leftrightarrow \vec{x}_i = \vec{x}_j$$
$$\delta_{i,j} = \delta_{min} \Leftrightarrow \vec{x}_i = \vec{x}_j \forall \vec{x}_i, \vec{x}_j \in X$$

5. Semantics. The semantic of $s_{i,j} > s_{i,k}$ is that object i is more similar to object j than is to object k. The semantic of $\delta_{i,j} > \delta_{i,k}$ is that object i is more dissimilar to object j than is to object k.

8 See Further Work.

9 Mercer's Conditions says for which kernels exists a pair (Euclidean Space, mapping function) with the properties described. (We didn't described much, again for more information [2])

An index that satisfies this properties is a proximity index (similar or dissimilar).

Another interesting property is:

$$s_{ij} = s_{max} \Leftrightarrow \vec{x}_i = \vec{x}_j = \vec{x}_k \qquad \forall\, \vec{x}_i, \vec{x}_j \in X$$

That is, two objects are regarded as more similar, the more similar they are with one another and with reference to a third “ideal” or prototypical object.

Finally, a space X where a proximity index $(p_{i,j})$ has been defined forms a semi-metric space, denoted (X,p), being p either s or δ .

So we have usually two spaces: the input space and the feature space. Usually SVM work with Kernels, the Kernel requires the feature space to be an inner product space. But when using a similarity function it is not required any projection so there is no feature space.

Both kernels and similarities work with points in the input space. If we can define a proximity index with that points we can say that the input space is “semi – metric” and that allows us to use a similarity function. That similarity function will not be a “masked” inner product.

A good thing about similarities is that it is not required for them to be “Positive-definite”[10] (as Kernels are required to), this means that we have a bigger semantic freedom to define what our similarity will do.

About Mercer's condition, [4] says (4.1) that even for kernels that do not satisfy Mercer's condition (and we don't know if it satisfies or not), one might still find that a given training set results in a positive semidefinite Hessian[11].

Can SVM work with a similarity function that would return descriptions instead of numbers? It should.

That descriptions should define features and our space has n dimensions (one dimension per feature); so given the description from a similarity function we should be able to allocate it in our space. Once everything is allocated we should be able to draw an hyperplane that would separate it in classes. It is easy to say, it is hard to implement. The core idea is the same that is using SVM: find an hyperplante. The formulation would probably not be the same.

Our objective is not to make the mathematical formulation easier, our objective is to make the problem easier from a practical point of view.

10 That is good and also bad. Being “positive-definite” means that the surface of the function to minimize is convex, this means that the problem has only one solution. This will not happen with similarities and we may have some local minimums, but if one local minimum is good enough that should be enough for us to solve the problem.

11 The square matrix of second-order partial derivatives of a function; that is, it describes the local curvature of a function of many variables.

4.2 Why using a Similarity function instead of a Kernel?

Because it finds a solution in a human state and because we know how to solve most of the problems so we don't need to reformulate our solution to make the machine solve it. We can use our own knowledge. That's artificial intelligence. The bad point is that a kernel matrix can probably solve more problems than a similarity function. We don't say: stop using Kernels, use Similarity. We say that there are some problems more suitable for our solution.

When someone reads books or papers about SVM all the problem they try to solve are about maths because they are math people. They usually talk about points in spheres with some radius or points generated by some statistical distribution... that kind of problems can't be solved with a similarity function. A similarity function can solve real problems; the type of problems someone who is working in DM with real world data has to solve like classifying documents, images, blood analysis, weather conditions... all these problems have a human solution because they were solved before computer existed (or before computers were so cheap that everybody could use them).

We say: you don't need to project it to some space, to use statistical values,... the human knowledge is enough for human problems.

This new formulation provides a new way to get the same solution. The good point is that it is easier to define a similarity than defining a kernel, the bad point is that kernels work with less work because the environment is prepared for them. In most cases actually is not necessary to create a kernel. The already done kernels like “Gaussian kernel”, “Chi squared kernel”,... would solve most of our problems. We only need to convert our input to some range of real numbers and that is usually a very easy task. In the chapter before we proved how that works.

A user can choose between converting everything to real numbers and let the magic flow or create a similarity function and have a bigger control about the process.

Showing a new way to work with SVM is not our main objective, that is actually a side-effect: using a similarity function makes easier finding a solution to the problem where this thesis started. The problem we will try to solve in this thesis is about finding an easier and better way to work with heterogeneous data. We achieve that with similarity functions and we will show how in the next pages.

4.3 What is heterogeneous data?

As computer people when we think about a problem we think about a real world problem; math people use to not care about this kind of problems because they find them easy. It is easy to find a math theorem about how to solve “weather prediction[12]” or “how to predict

12 “Weather prediction” in a DM context. Provided some data to predict the future. Emulate the Earth requires the most powerful computers. That is not an easy problem. It is probably more related with “Simulation”.

what the people buy in supermarkets[13]". Sometimes they call it "toy examples": easy examples they use to show the surface of a theorem to people like me. Math problems are usually like "classifying sequences of numbers", "classifying physic properties", "classifying statistical distributions",...

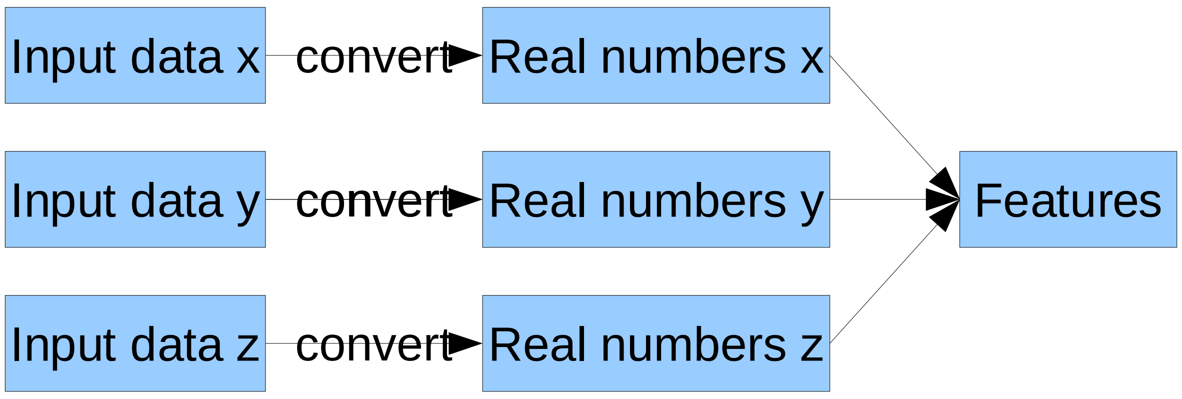
But we know that predict what the people buy in supermarkets is not as easy as it seems. While the math solution is easy getting a real good solution is not because we have to struggle with some problems they don't care about. One of them is heterogeneous data, real world problems require different types of data. Our supermarket problem is not only about the most popular products; there is also some other useful information like the information related to the buyer: sex, age, money, studies, also time he/she spent in the supermarket, schedule,... More information can be related like: which products were cheaper, which were on sale, preferred brands,...

Another example about heterogeneous data is predicting the movement of a robot. The robot is receiving information from different channels: sound waves, pictures, position of other robots, weather conditions, the map,...

A usual example is related to medicine when a doctor has to predict the sickness by some different inputs: blood analysis, ray-x images, what the patient feels,...

Dealing with heterogeneous data has usually been a problem without a proper solution in DM and this kind of data is very common in real world problems. That is why a better solution is needed and that is what we try to show here.

What is usually done is: convert all the data to the same type (real numbers), work with it. So we would need to convert the ray-x images, the blood analysis and the description given by the patient and throw everything to SVM. That uses to work. But in that step is easy to see that we are losing a lot of information.



This is exactly what we did in chapter 2. We converted our data (strings) to real numbers. We focused on if a word appeared or not, its frequency,... with real numbers it was easy to create features and finally send them to SVM. We were only focusing in morphemes and we lost all the other information related with documents.

13 As before. The human behavior is not usually an easy problem. If we go that deep it would be more related with psychology, not with DM.

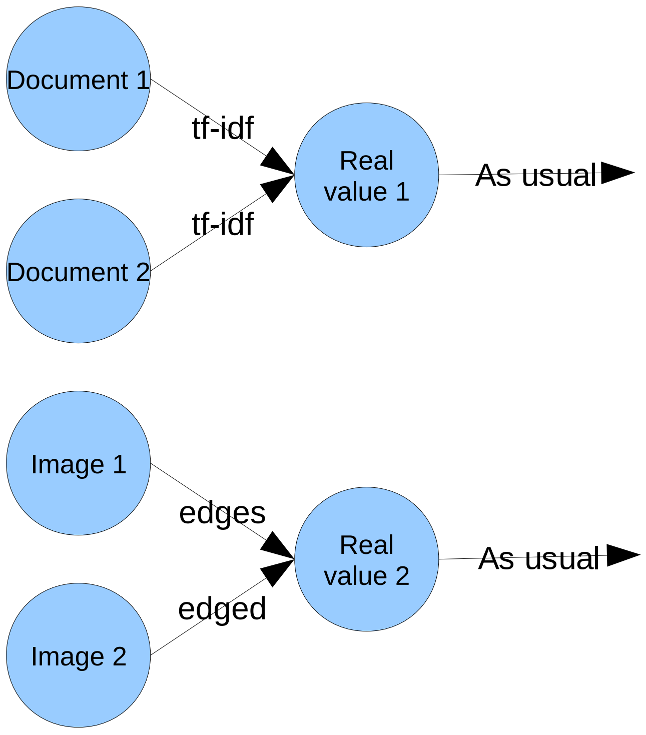
In case we have different kinds of data the process is the same, although is not that easy to create the features. We have two solutions:

- Put all our work in finding a way to represent our data using numbers and later throw everything to a working kernel (like Gaussian). [Bad solution]
- Create a new kernel that projects the data to a working feature space. “Finding a way to represent our data” is usually part of this projection. [Good solution]

4.4 Our solution

At first we focused on Neural Nets. We thought about adding a new input layer where each neuron on that input layer will have a different type (based on the input data). The next neuron's layer would compute their input with a different calculation rule based on the input layer. The rest of the neural net will have the normal behavior.

If our data is about images and documents our first layer will contain some neurons that will store images and some neurons that will store documents (instead of storing a real number, which is the usual data stored by neurons). The input of the next neuron instead of being a linear combination of its inputs applying later a sigmoidal function (this is one of the multiple ways to calculate it) will calculate the neural value related to the type of input it has. If the input is a document, for example, it may calculate the “tf-idf”.



The above image shows the basic idea (of course more calculation is required apart from “tf-idf”, this is a simplification to show the core). We started to work on that solution but it would require some big changes about Neural Nets. For example if we use

"Backpropagation", how can we do that last step when returning? How can we compare results? Is our function differential? How? There may be a solution but it was not easy to find it.

Later we found that all the problems we had when working with Neural Nets were not there if we used a SVM. Instead of adding that extra layer we only need to add a similarity function as explained before.

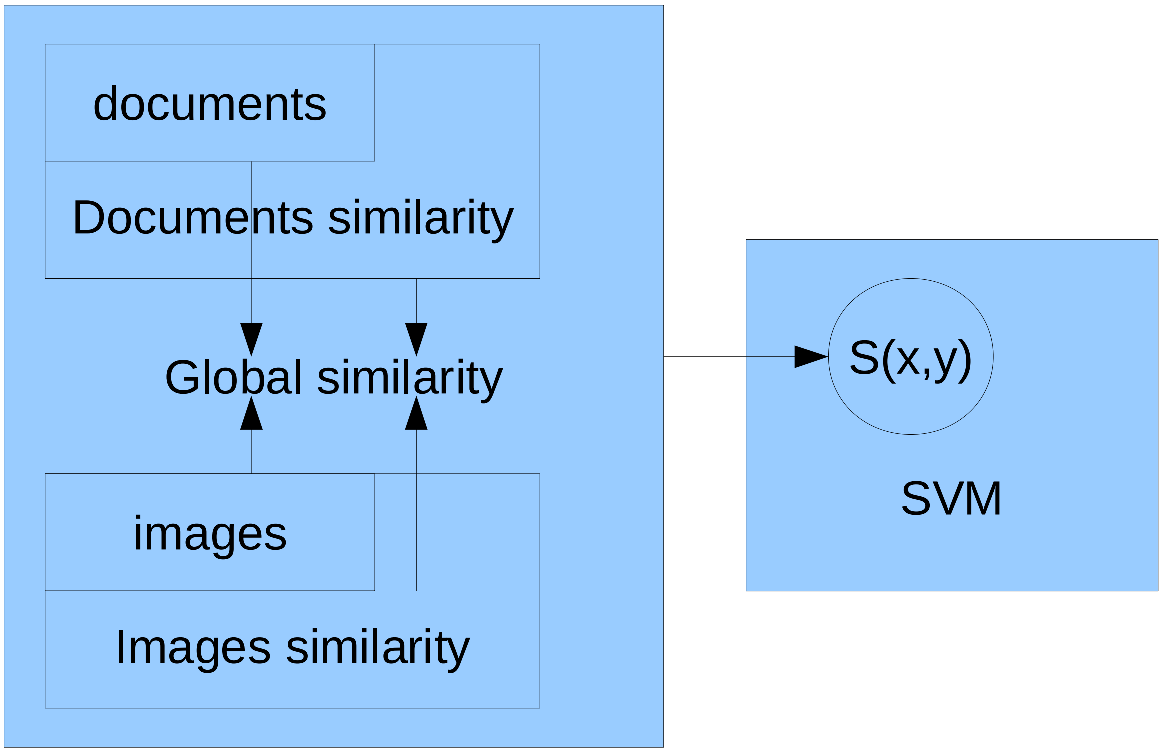
So the solution is: if you have to deal with different types of data you should provide a similarity function for all of them (there are known similarity functions for documents, images,... and we can easily create new ones). How to mix all that similarity functions? With a global similarity function.

We will use as example face descriptions and face images. We know how to say if two face descriptions are similar and we know how to say if two face images are similar. Now we have to define the new global similarity. We have to use our human knowledge, it can be something like:

- If document A and B are talking about eyes, return the similarity between image A and B in the region where the eyes are.
- If image B doesn't have a mouth (the image can be cut) return the similarity between documents A and B were they talk about mouth (this is a new side effect to deal with missing data).
- If hair in image A is yellow and hair in image B is black, return a 0 (so global similarity is not only dealing with similarities, it can check also directly the data).

And so on...

This way of working with heterogeneous data is easier and we are not losing information because we don't need to convert the input data to anything. We only have to define how the similarities will be, nothing more, and the problem is solved.



4.5 Other (new) solutions

While we were working on this solution (by working we mean to wait for me to finish the mandatory courses and do the final thesis, everything started around three years ago) the DM community focused on solving the way to work with heterogeneous data. They provided a good solution called MKL (multiple kernel learning).

When we said to use a different similarity function for each type of data they use a different kernel for each type of data.

When we define a global similarity function to solve the problem they try to find the best linear combination within all the kernels. They combine the kernels and try to find a parameter that will say how important each kernel is to find the solution.

MKL are better in the sense that they don't need an expert, we only need to define the different kernels and MKL will find the best solution for us. Our solution is better because we can define more complex ways to mix the results than a linear combination and that would give better results. The example we explained before with images and documents about faces would be impossible in MKL. Our solution is also better in the sense that we don't need to define anything new and that everything that worked with simple kernels will work with similarity functions. It is said that some MKL can achieve this too [7].

As showed in the “image-descriptions” quick example our solution also is useful when dealing with missing data. Missing data is probably the other big issue in DM when working with real world examples.

4.6 Implementation

My personal objectives for this part are:
- Learn better C++ [17]
- Understand the Shogun Library as a developer

Our idea has been shown; the objective now is to show it working. As said in the chapter before we decided to work with Shogun. Shogun is a +80K lines library, it has around 200 classes and 9 active developers. The project started nearly three years ago and the last update was one month ago. It is a powerful DM tool that focuses on SVM (it has other several methods) and it specially focuses on MKL because Shogun's developers are part of it. MKL is related to what we try to solve.

The first step when developing software is to study the facts. It requires a couple of hours searching through the library website, reading the author's pages, checking if it is outdated, if it has an active community working behind, checking the quality of the code...

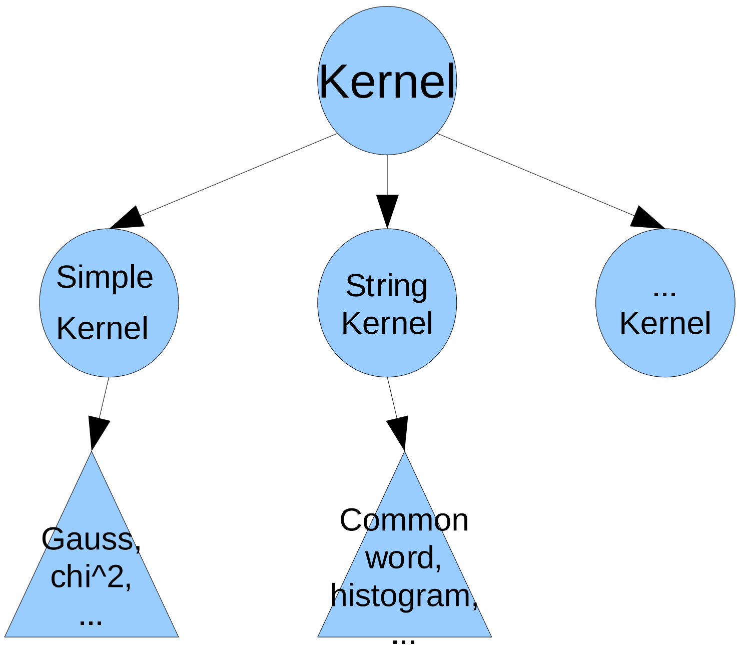
After studying Shogun's facts one thing was clear: it must provide most of the things we need because we are trying to solve the same problem. We don't need to write much code because it may already be written by them, it is a matter of studying the code. We need to adapt our theory to their library. We have to design how to do it and we have to use the two most important object oriented ideas: polymorphism and inheritance. The most code we write the most our design is bad. Inherit classes, rewrite only required lines.

We don't consider ourselves good coders while Shogun's developers are quite good, the less we write and the most we use their code, the better. We spent most of the time in our implementation reading the Shogun's library, collecting tools they did, making designs to their tools, reading again, simplifying the design.

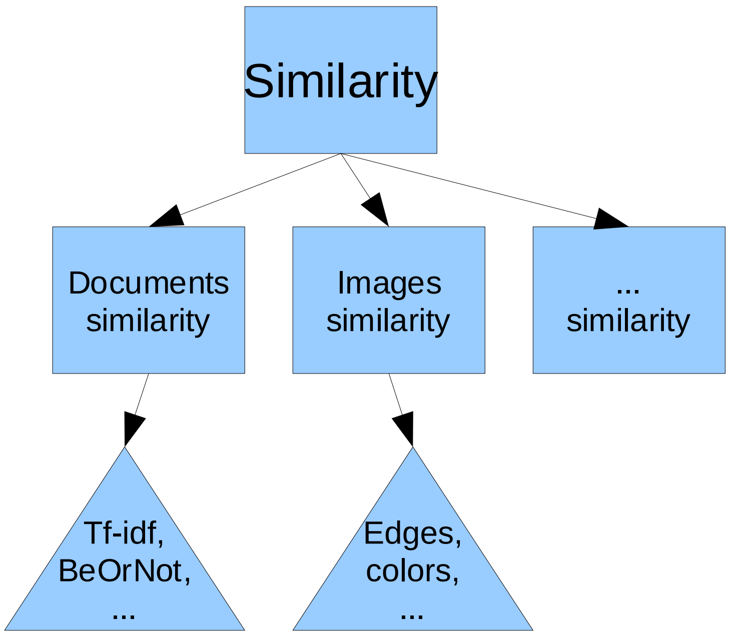
4.6.1 Implementing similarities

At first we focused on implementing the concept of similarity. Once this is working we will make it work with heterogeneous data. Our first solution was the direct one: the design that would make someone that has not read deeply the library.

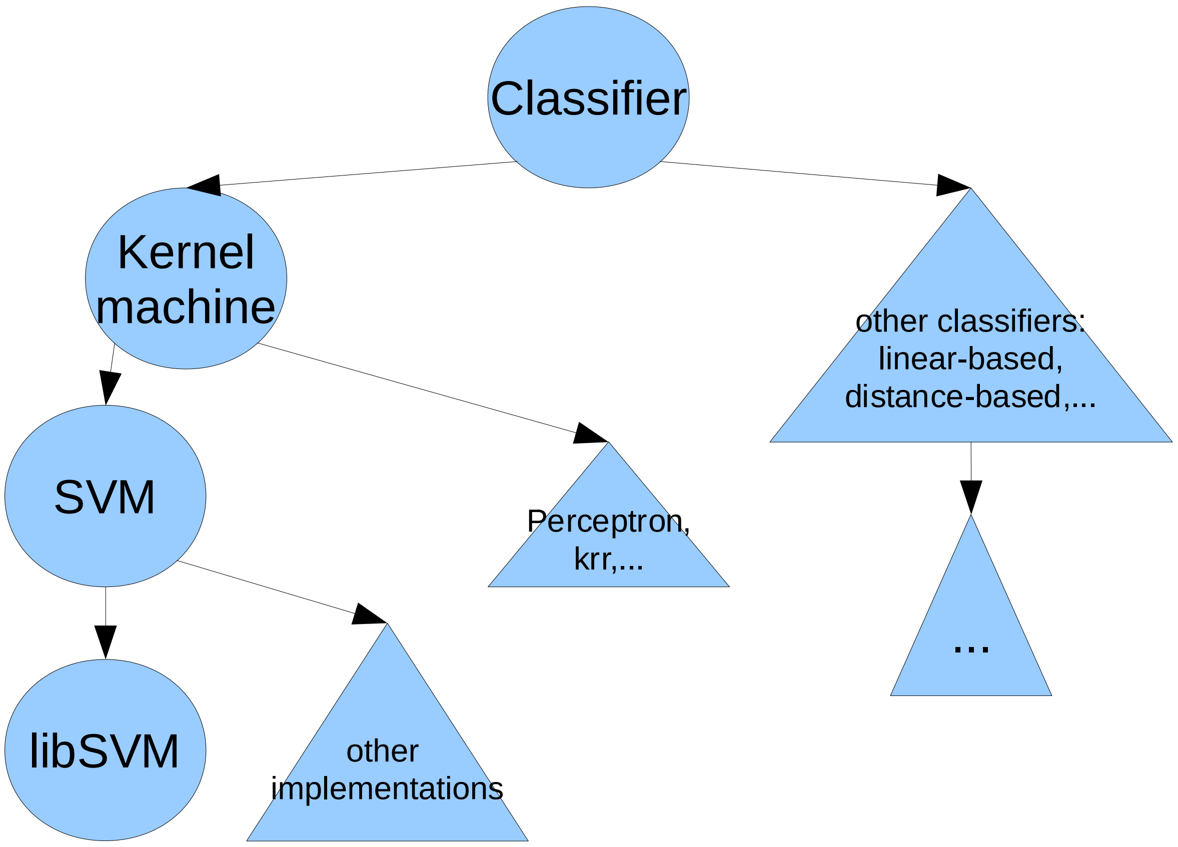
From now until the end in the class diagrams a square will mean a class owned by us while a circle will mean a class owned by Shogun. So squares are modifications. Finally, it is a convention that in C++ class names start with a “C” like “CPerson”; the next diagrams are not a C++ class representation, they are a design representation. That is why we will call classes by they name so “CPerson” will be “Person”.



Shogun has a class called Kernel. That class has a lot of children defining a lot of different kernels. The class Kernel contains almost everything and each layer is specializing the layer before: inheritance. If we wanted to use a Similarity function where a Kernel is we have to create the same kind of hierarchy. The class "Similarity" will be at the same level as "Kernel".

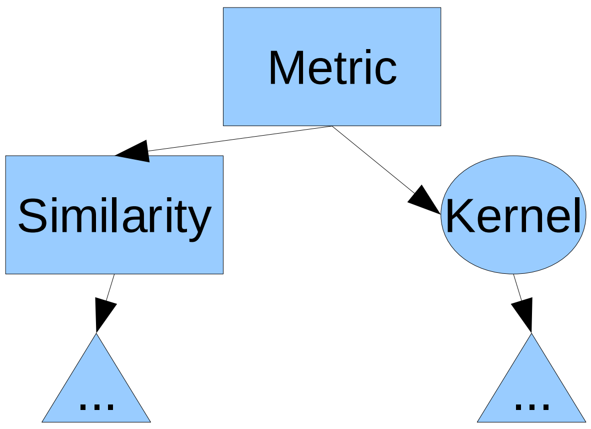


The design seems simple. The problem here is that the Kernel Class (the basic parent) has around 2000 lines and it is doing a lot of things we also have to do with our Similarity Class if we want Shogun to be able to use it. The same can be applied to Kernel's children. Anyway that can be solved with some copy/paste and luck. The real problem about this implementation was that we also needed to change the SVM tree.



Everything from “Kernel machine” class should be fresh new code. That means several thousand of new lines not only defining SVM with similarities but also making everything workable with Shogun's library. That would mean to redefine part of the core of Shogun, like 30% of what is already done during years by good developers. This design was not an option, we must read the library carefully and use some of what it is already done, not only the “classifier” base class.

The next design was as follows:

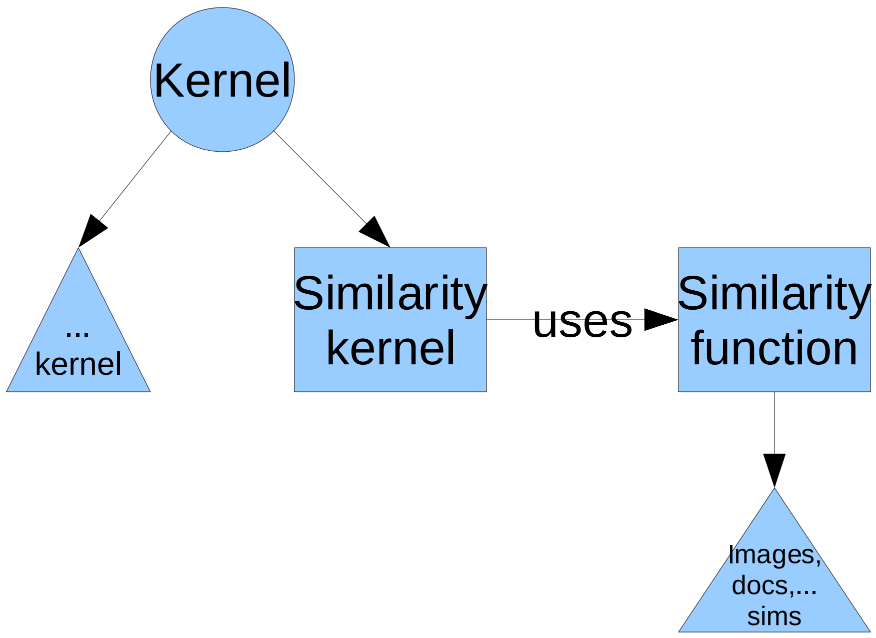


The idea was to create a new class called Metric that will contain both Similarity Class and Kernel Class (both classes with a tree as showed earlier). We still have to do some work here, actually a bit more because we will need to rewrite some lines in Kernel tree and create all the Similarity tree, but this new design will allow us to not having to define anything new in the SVM tree because using the already done code where SVM was waiting for a Kernel object we would send there a Metric Object. With this design only some lines would have to be rewritten in case SVM is expecting the results from a Kernel or from a Similarity. This design is far better, but not the best because too much work is still required, specially in the SVM part.

When studying Shogun and different designs one important problem arises about Similarity functions that will also be the key for the final (and best for us) solution. A Kernel is a matrix. It is usually filled quickly only doing the inner product between feature vectors, later when SVM start to train it is only grabbing the results from that matrix. We know that accessing an array is in most languages a very fast operation. That access is done millions of times.

A Similarity is a function so each time SVM needs some results it has to call the similarity function between two training points. Imagine having to calculate a function value millions of times. What is worse is that the same results have to be calculated several times. It is obvious that we need a caching system and by the definition of our problem the best structure is a matrix, exactly the same as Kernels.

Why not to create a new Kernel called “Similarity Kernel” that will have a similarity function to calculate what kernel does with inner products? We only need to create a new kernel, make it to store a function (a similarity) and change how it calculates its values to call that function. We only need to rewrite one line: the calculation line. Everything else will work out from the box.



In our code “Similarity function” is called “Similarity”. It is handmade; it is a class that defines how to compute the similarity between two training points. It has to return a real value. “Similarity kernel” is a son of Kernel. It defines two methods: the creator and the compute. The creator stores the similarity object, the compute calls the similarity object for its output. The similarity accepted by “Similarity Kernel” can be anything.

```
1. CSimilarityKernel::CSimilarityKernel(int32_t  size,  CSimilarity* sim)
2. : CKernel(size), similarity(sim)
3. {
4.       ASSERT(similarity);
5.       SG_REF(similarity);
6. }
7. # SimilarityKernel.cpp
```

This is the creator of our class (in the header file it is declared as CKernel's son), we only require two parameters: size and similarity. Size is something that will be managed by CKernel. We assign the similarity object provided to our attribute called “similarity”. Line 5 is required by how Shogun manages references. We will not go deep as how to develop to Shogun or how the library works. That would require dozens of appendix's pages and Shogun itself has a very good documentation. It is enough to say that in the class destructor we will use a function called “SG_UNREF”.

```
float64_t CSimilarityKernel::compute(int32_t idx_a, int32_t idx_b)
{
      float64_t result = similarity->similarity(idx_a, idx_b);
      return result;
}
```

```
# SimilarityKernel.cpp
```

That is how we fill our matrix: getting the results from the similarity function. Variables “idx_a” and “idx_b” are the indexes to the current training points. This is how Shogun works: the similarity function will be the responsible to grab the real data using that indexes. The class has more code (between .cpp and header file around 150 lines) required by Shogun's but that's the core. We will never show the full code, only what is required to follow our design.

The similarity class (CSimilarity as seen in the first piece of code) requires more classes: first of all the “Similarity.cpp/h” base class (around 600 lines of code, we will not put that code here, it is mostly code required by how Shogun works). Then we wrote different templates [14] to define different kinds of similarities. In our case we created one called “Single” to work with a similarity that will use numbers, one named “String” to work with a similarity that will use strings, one named “BagOfWords” to work with a similarity that will use lists of strings (similar to what we get after tokenizing a document) and finally a similarity called “Combined” to combine different similarities; we will talk about this one later.

From any of these templates we can create new similarities. Usually only the “compute” function is required to be rewritten (in case one needs to work with different data a new template must be created). For example from “SingleSimilarity” we created a similarity called “DocIntersectionSimilarity”. This similarity receives a list of numbers (0's or 1's as in the examples we worked before) and computes the intersection between both documents. The compute function is as follows:

```
float64_t  CDocumentIntersectionSimilarity::compute(int32_t  idx_a, int32_t idx_b)
{
      int32_t alen, blen;
      bool afree, bfree;

      float64_t* avec=
            ((CsimpleFeatures<float64_t>*)lhs)->get_feature_vector(idx_a, alen, afree);
      float64_t* bvec=
            ((CsimpleFeatures<float64_t>*)rhs)->get_feature_vector(idx_b, blen, bfree);

      ASSERT(alen==blen);

      float64_t result=0;

      for (int32_t i=0; i<alen; i++)
      {
            if ((avec[i] > 0) && (bvec[i] > 0) ) {
                  result++;
```

14“template” as used in C++

```
19.             }
20.         }
21.
22.         ((CSimpleFeatures<float64_t>*) lhs)->free_feature_vector(avec, idx_a, afree);
23.         ((CSimpleFeatures<float64_t>*) rhs)->free_feature_vector(bvec, idx_b, bfree);
24.
25.         return result;
26. }
27. # DocumentIntersectionSimilarity.cpp
```

We compute the similarity in lines 15-20. It is doing the intersection so it is quite easy. All the other lines are how Shogun works: from the indexes we grab the real data and later we free it.

4.6.2 Similarities or Kernels

We talked a lot about how nice similarities are and now we are using a Kernel. We are not really using a Kernel, we are using a matrix to store the similarity results. By design it was the best to use what Kernel class offers to us. The Kernel itself is empty, we only use its matrix (and its name) to make everything easier.

If instead of using a Kernel we tried the first solution the results would be the same. From the point of view of the user he/she only has to define new similarity functions and use the similarity (empty) Kernel. Our Kernel is not projecting and is not calculating the inner product. Theoretically our “Similarity Kernel” can't be called Kernel, practically it makes everything easier.

4.6.3 Making it work with Python

End users are supposed to use Shogun with Python[15]. Each super-class in Shogun contains a configuration file to create bindings from C++ to the other languages (in the appendix there is a full example about how to run Shogun with Python). That configuration file was one of our hardest tasks because we had to modify some existing files and create news (for CSimilarity, for example). We talk about this because we spent here a lot more time than we predicted. It is the kind of slacks you get in software engineering cycles.

Our objective in this section is to show how easy is to work with similarities. Shogun is quite simple and powerful to use: with a dozen of lines you can have a complex DM algorithm running. We tried to make similarities as easy to use as in general Shogun is:

```
feats_train_0or1 = RealFeatures(array(train_set_0or1).T)
feats_test_0or1 = RealFeatures(array(test_set_0or1).T)
similarity = DocIntersectionSimilarity()
```

15 Shogun can also be used as an end-user in C++, R, Octave,.. but Python is the recommended way.

```
4. kernel   =   SimilarityKernel(feats_train_0or1,   feats_train_0or1,
   similarity)
5. svm = LibSVM(10, kernel, labels)
6. svm.train()
7. out = svm.classify(feats_test_0or1).get_labels()
8. #Python test file
```

A lot of code is omitted (that script has more than 100 lines) but that contains the essential. Lines 1 and 2 are creating the feature objects as required by any Shogun method. Line 3 is creating a similarity, that similarity must use the same kind of features we declared before, in this case "RealFeatures" otherwise it will not work. Line 4 creates our kernel (sigh). As we said this is not really a kernel: it is a matrix storing the results from our similarity function using the same name. The kernel requires three parameters: the features to train[16] and the similarity to calculate the matrix. Once this information is provided we start the SVM process, line 6 is doing the hard job here. Line 7 is the output from new points.

The only new thing we have to do is to create a similarity object while all the other code is the usual code when working with Shogun. We only need one new line and the process is intuitive as it is when working with Shogun.

To check the output we used the accuracy as in the chapter before. The results were around 0.80 in all the similarities we tried[17].

An 80% is not a better result that the ones offered in the chapter before, it is actually quite similar. Our objective is not to provide a better way to work with documents (intersection and union are the most basic similarity calculations that can be done with documents). Our objective is to provide a different way to calculate it. An 80% is enough to prove that our idea is working and that it is not just random luck. We are sure that with proper similarities function the results can be better that the ones offered by the common kernels.

One interesting thing is that if someone checks the matrix after using a Kernel the results there will be very cryptic because they come from "complicated formulas" and it is hard to read it and to understand it.

If someone reads the matrix after using a Similarity function it is easy to read. It will contain (in our case) integer numbers representing how many words the document share (intersection, for example). We can check which are the support vector and understand why. Reading our kernel we can also improve our results or we can know why we are failing. We can understand every step while Kernels are a black box; they are not providing any more information than the solution in case we are not experts on maths or statistics.

While a Kernel tries (usually) to emulate an statistical behavior a similarity tries to emulate a human behavior, that's AI. We make all the process "humanable".

16 Rows and columns.

17 Intersection/union of documents represented by numbers or strings.

4.6.4 Combining similarities

All that we have explained until now about the implementation was about using similarities instead of kernels. As we said in the theoretical part that is not our main objective, it is the key concept about how we are going to solve our main objective: deal with heterogeneous data.

It is interesting how the “idea's path” went from A to B and the implementation went from B to A. The path started with: “We have to find a better way to work with heterogeneous data”, later we said: “each kind of data has to have its own method to classify it, that method must be part of the DM algorithm, not a pre-process”. Finally we said “that method will be a similarity function, we will use this with SVM”.

In the implementation we started by making SVM work with similarity. Now we will try to combine different similarities and in the next step we will make it work with the heterogeneous data.

Similarity is the key concept. It was the hardest part to design and implement and now that everything is working it provides an easier framework to finish the rest.

In the previous example we were using “SingleSimilarity”: in our example “DocIntersectionSimilarity” is a “SingleSimilarity”. We also prepared the templates to combine similarities and from that templates any kind of similarities that require a combination of them can be created.

SimilarityKernel doesn't care whether the Similarity function we provide is Single or Combined or whatever. It has to be an object from the Similarity tree (inheritance) and it has to provide a function called compute (polymorphism) that will return a number as value. So when we will have to create combined similarities nothing has to be modified: we only have to extend our code.

One of the combined similarities we created is called “DocumentCombinedSimilarity”. It will calculate the Tanimoto Similarity [15], which is:

$$\textit{Tanimoto Similarity} = \frac{d_1 \wedge d_2}{d_1 \vee d_2}$$

Given two documents their intersection over their union. We may have a similarity to calculate the intersection between documents and another to calculate the union between them. We use a combined similarity that will... combine the results:

```
float64_t    CDocumentCombinedSimilarity::compute(int32_t    idx_a,
int32_t idx_b)
{
    float64_t  result1  =  intersectionSimilarity->similarity(idx_a,
idx_b);
  float64_t result2 = unionSimilarity->similarity(idx_a, idx_b);
  float64_t result = result1 / result2;
  return result;
}
# DocumentCombinedSimilarity.cpp
```

As can be checked we are using a compute function, the same function used before (with different code). This code is doing the obvious (and expected): grabbing the intersection and the union and returning one over the other. It is simple but this way of combining the input data is much more powerful than most of the methods used today.

That simple division is something that MKL can't perform. That code can be anything, instead of a division it can be one hundred lines of code and instead of calling an intersection and union similarities it can call whatever; it may be computing 10 similarities at the same time... it can do whatever is required. We can also check that in that function we have direct access to the data, not only to the similarity results.

```
CDocumentCombinedSimilarity::CDocumentCombinedSimilarity(CSimilarity* sim1, CSimilarity* sim2)
: CCombinedSimilarity<float64_t>(), intersectionSimilarity(sim1), unionSimilarity(sim2)
{
  ASSERT(intersectionSimilarity);
  SG_REF(intersectionSimilarity);
  ASSERT(unionSimilarity);
      SG_REF(unionSimilarity);
}
#DocumentCombinedSimilarity.cpp
```

The creator is extending the “CCombinedSimilarity” template (in this case the template is typed to floats). The only thing this subclass is doing is storing the similarities that will later be used in the compute function. DocumentCombinedSimilarity is probably a too wide name that can create confusion because this similarity can't combine anything to calculate how similar two documents are. A better name is probably “CDocumentCombinedTanimotoSimilarity”.

It requires the first similarity to compute the intersection and the second similarity to compute a union. The header file stores that similarities as “CSimilarity”:

```
CSimilarity* intersectionSimilarity;
CSimilarity* unionSimilarity;
# DocumentCombinedSimilarity.h
```

This means that these similarities can be anything (single, combined,...). So we can use combined similarities to calculate a combined similarity and so on.

What follows is a Python example showing this working:

```
feats_train_0or1 = RealFeatures(array(train_set_0or1).T)
feats_test_0or1 = RealFeatures(array(test_set_0or1).T)
similarity1 = DocIntersectionSimilarity()
```

```
similarity2 = DocUnionSimilarity()
similarity = DocCombinedSimilarity(similarity1,similarity2)
kernel = SimilarityKernel(feats_train_0or1, feats_train_0or1, similarity)
svm = LibSVM(10, kernel, labels)
svm.train()
out = svm.classify(feats_test_0or1).get_labels()
#Python test file
```

We only have to create the sub-similarities (single or combined), the combined similarity and give that last one to the kernel. The kernel will call the compute function and that tree of similarities will provide the result. The user only has to define how that compute function will be in all the cases and as we proved inside that code anything can be done. If a Turing Machine that writes 1's and 0's in a tape has been proved to calculate anything a computer can a C++ function can too.

That script using the combined similarity has an average accuracy of 85%. Enough to say that we can combine similarities to make SVM work.

4.6.5 Finally: heterogeneous data

Idea's design went from A to B. Implementation went from B to A. We have arrived to A again. We have created all the tools we need and they have proved to work. SVM worked with similarities and we said how to combine them; SVM worked with combined similarities.

Before, we said: "each input will have its own similarity". That's what we did before with combined similarities but there is still one last step because we were using the same kind of data for all the inputs: all the similarities we have been working until now worked with the same feature vector, over the same positions. Since Shogun is prepared to work with MKL it has to have a way to work with heterogeneous data. Shogun provides the perfect tool we need: CombinedFeatures.

CombinedFeatures is a class that will contain all the data we need. It doesn't care about the kind of data because it can contain at the same time documents, images,... Now it is time to think how to design a solution given that tool.

After reading carefully again Shogun's code paying attention to how they work with MKL we decided that each similarity will keep care of its own features (they do something very similar). It was enough to add the following attributes to CSimilarity (that will be inherited by all the sub-similarities):

```
/// feature vectors to occur on left hand side
CFeatures* lhs;
/// feature vectors to occur on right hand side
CFeatures* rhs;
#Similarity.h
```

The type is Cfeatures and it means that it can contain anything: documents, numbers, images,... combined features too.

We decide that it will be CombinedSimilarity's responsibility to provide each sub-similarity with the features required:

```
bool CDocumentCombinedSimilarity::init(CFeatures* l, CFeatures* r)
{

  bool result=CCombinedSimilarity<float64_t>::init(l,r);
  ASSERT(l->get_feature_class()==C_COMBINED);
  ASSERT(r->get_feature_class()==C_COMBINED);
  ASSERT(l->get_feature_type()==F_UNKNOWN);
  ASSERT(r->get_feature_type()==F_UNKNOWN);

  CFeatures* lf=NULL;
  CFeatures* rf=NULL;

  CListElement<CFeatures*>* lfc = NULL;
  CListElement<CFeatures*>* rfc = NULL;

  lf=((CCombinedFeatures*) l)->get_first_feature_obj(lfc);
  rf=((CCombinedFeatures*) r)->get_first_feature_obj(rfc);

  result = intersectionSimilarity->init(lf,rf);

      if (!result) {
        SG_INFO( "Initialising first sim failed\n");
  }

  lf=((CCombinedFeatures*) l)->get_next_feature_obj(lfc) ;
  rf=((CCombinedFeatures*) r)->get_next_feature_obj(rfc) ;

  result = unionSimilarity->init(lf,rf);

  if (!result) {
        SG_INFO( "Initialising second sim failed\n");
  }
#CdocumentCombinedSimilarity.cpp
```

We can see that CombinedFeatures is a list of features so it iterates over the list sending to each similarity its corresponding features. In this way each similarity can work with different kind of features. Also some similarities can work over the same set. We have a total control over it. This code can be done with a loop easily but we preferred to show it without a loop to make clear how each similarity is receiving its features.

We just showed how to solve our initial problem, what follows is a Python working example:

```
1.  # Get the features RAW, they are strings
2.  featuresets_raw = [' '.join(set(document_features_string(d))) for
    (d,c) in documents]
3.  train_set_raw,test_set_raw= featuresets_raw[300:1000],
    featuresets_raw[:300]
4.  feats_train_raw = StringCharFeatures(train_set_raw, RAWBYTE)
5.  feats_test_raw = StringCharFeatures(test_set_raw, RAWBYTE)
6.
7.  # Get the features 0 or 1, they are number
8.  featuresets_0or1 = [document_features(d) for (d,c) in documents]
9.  train_set_0or1, test_set_0or1 = featuresets_0or1[300:1000],
    featuresets_0or1[:300]
10. feats_train_0or1 = RealFeatures(array(train_set_0or1).T)
11. feats_test_0or1 = RealFeatures(array(test_set_0or1).T)
12.
13. combined_feats_train=CombinedFeatures()
14. combined_feats_test=CombinedFeatures()
15.
16. combined_feats_train.append_feature_obj(feats_train_raw)
17. combined_feats_test.append_feature_obj(feats_test_raw)
18. combined_feats_train.append_feature_obj(feats_train_0or1)
19. combined_feats_test.append_feature_obj(feats_test_0or1)
20.
21. #this similarity works with strings
22. similarity1 = DocIntersectionStringSimilarity()
23. #this similarity works with numbers
24. similarity2 = DocumentUnionSimilarity()
25. similarity = DocumentCombinedSimilarity(similarity1, similarity2)
26.
27. kernel = SimilarityKernel(combined_feats_train,
    combined_feats_train, similarity)
28. #kernel.get_kernel_matrix()
29. svm = LibSVM(10, kernel, labels)
30. res = svm.train()
31. out = svm.classify(combined_feats_test).get_labels()
32. # Python final test example
```

The first lines, from line 2 to 5, are extracting features from documents (d is a list of documents). The features extracted are strings. It is just saving each word as a feature in its string format. That similarity will receive is a list of strings.

Lines from 8 to 11 are doing the classic conversion from documents to 0's or 1's depending if that word appears or not.

Lines 13 and 14 are creating the objects CombinedFeatures for training and test. Later, on lines from 16 to 19 that objects are filled. We can check that it is adding both the list of strings and the list of numbers. Heterogeneous data.

From line 22 to line 25 we are creating the similarities as we did before. Line 27 is creating a SimilarityKernel as we did in the very beginning; the difference now is that the features are combined. Nothing else is changed: the code from SimilarityKernel is the same as in the first example.

This is a “silly toy example”. It is extracting two different kind of features from the same dataset, but this example is enough to prove that it works. It is easy to see that it can be adapted to bigger problems where each feature comes from a different source.

Also, what it is very important, we are using SVM without any change. It doesn't matter if we are working with heterogeneous data or combined similarities or single similarities: libSVM will work as usual because our design provided the matrix and the compute function. As we said in theory we just showed that our new method will not require any change to SVM algorithms. Not one, nothing.

<u>That example had an accuracy around 85%. Enough to say that it worked and that we proved a new working way to work with heterogeneous data.</u>

Finally we can also check that “DocumentCombinedSimilarity” is working as before. No change there was done. It is irrelevant for this class the kind of features the other similarities are using.

4.6.6 Some extra work was required...

In the last example we saw that we used “StringCharFeatures” to represent the documents as strings. “StringCharFeatures” is a feature provided by Shogun and it was not created to work with lists of strings: it was created to work with long single strings like DNA sequences. That required the similarity to do some extra work.

The result was that the execution was very slow. Just one execution took around 20 minutes so we needed to create a new kind of features that would perform better with our problem. We created the feature type “BagOfWords”.

This is where we lost most time coding and where we spent more time implementing it. Something that we didn't expect to do at the beginning was by far the hardest part. We still had to use “StringCharFeatures” to get the information from the “outside” and then tokenize it to our bag.

```
feats_train_raw = StringCharFeatures(train_set_raw, RAWBYTE)
feats_test_raw = StringCharFeatures(test_set_raw, RAWBYTE)

feats_train_bw = BagWordsFeatures(feats_train_raw)
feats_test_bw = BagWordsFeatures(feats_test_raw)
#BagOfWords example
```

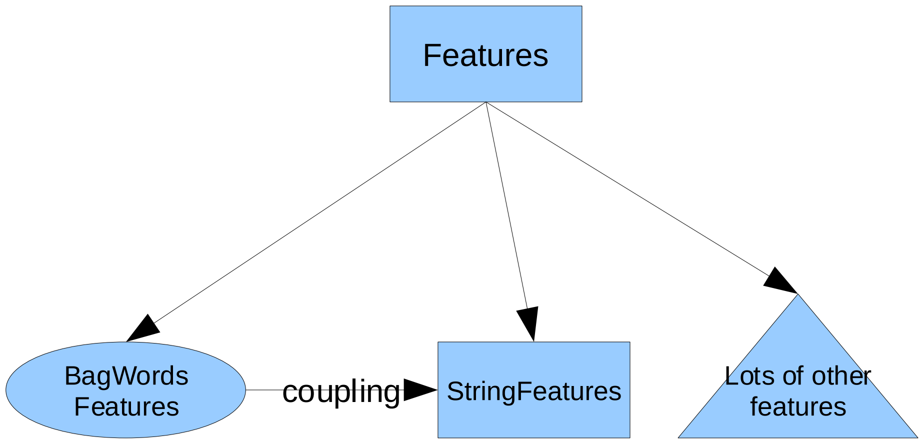
The rest of this script is the same as before but using the right similarity that suits with “bag of words”.

One thing we decided was to use in that new feature the structure “std::tr1::unordered_set”. Although it is not 100% compatible the “tr1” library comes with most of the system and “unordered_set” makes the execution even faster. It has to be noted that it is still slower than working with number.

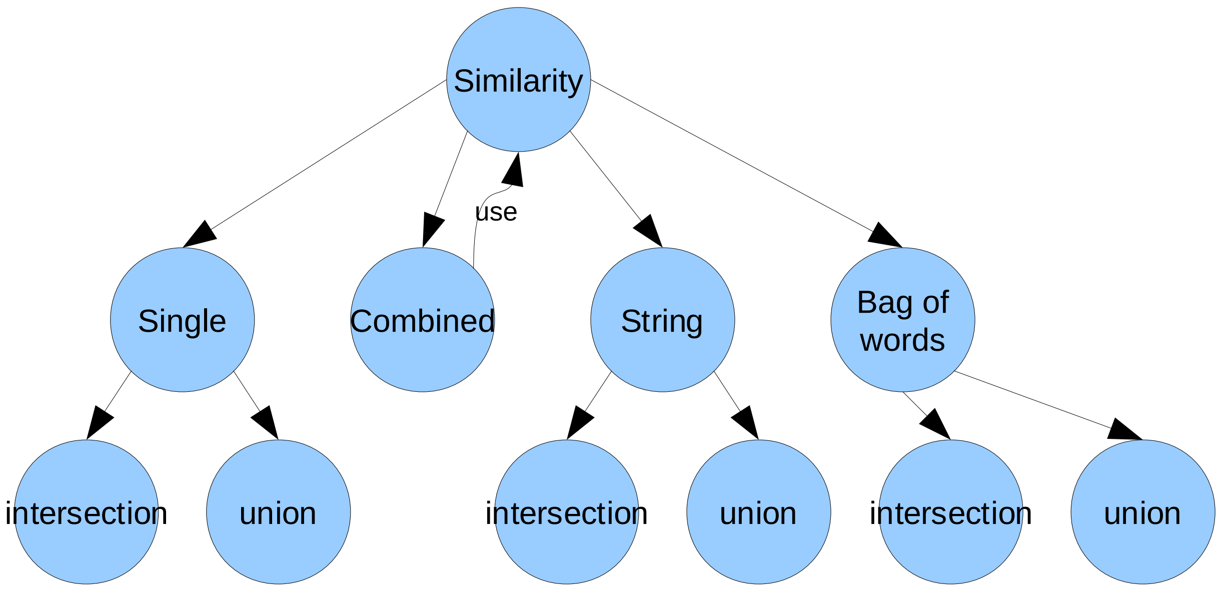
4.6.7 Complete Design

What follows is a some kind of class design, but more like it would be drawn in paper not in UML because the objective is to show how we integrated our ideas with Shogun. We don't focus on attributes or functions. The idea is to do a simple diagram as done in Shogun's documentation. A circle means our class, a rectangle means a class owned by Shogun, a triangle means more classes that we are not going to draw because they are obvious and in that way we would save some paper and focus the diagrams in the interesting things.

We started by the changes done to Features. To test heterogeneous we needed to create a new kind of feature, we called it “BagWordsFeatures”. We didn't had much time and We needed a solution, that is why we used an object from “StringFeatures”. That is a very bad OO decision generating “coupling”, but if I would have tried to create an independent new feature we would probably still working on it.



As explained why we added a similarity kernel, the final design was showed before. And we created some new similarities to work with, in the next diagram what is called just “Similarity” is the same as called in the diagram above “Similarity function”. Combined similarity needs at least one similarity to work with. I made an “use” relation; UML would probably call this kind of relation “composition”.



4.7 Comparing it with MKL

It is not easy to compare what we did here with MKL because what we did here is the work of one person for 4 months while MKL is the work of top scientists for several years. It is like trying to compare a Premier League team with a regional team from the University League. We say this because usually when you want to compare something is to provide better results, it is not the case since we only provided a toy example. We can't compare to MKL.

Theoretically it was explained before both approaches. MKL combines kernels, our method combines similarities. MKL needs to use a different kernel for each kind of data (actually each input) once that is provided, it will offer results. Our method needs to use a different similarity function for each input (something that is easier that creating new kernels) and then provide how to mix them (MKL does not need this step).

MKL works with all the kernels that are working nowadays, out method requires to write similarities from zero but that similarities already exists applied to other problems.

With MKL it is easier to get results (if you don't have to define new kernels) and it is faster, with out methods we are sure than you can get better results because it is a more powerful way.

Our method can actually be used with MKL because as explained we are using “kernels” to store similarities, we can use our similarities or combinations in MKL.

5. Conclusion

As usually happens with theses (especially with a 4 months thesis), the initial objectives[18] have to be simplified to be able to finish some work and present some results.

Each chapter was written in the same way we did it practically: from the basic idea until the final results. All the results and conclusions are spread inside the three big chapters so we will not reproduce all here again. We will talk about what we did and what we missed.

At first my knowledge about SVM was basic, I got the idea about “using similarities with SVM to deal with heterogeneous data” more or less, but I didn't fully understand what that meant. After some background work I understood what that really meant. That is explained in the introduction.

The next objective was to use Shogun and some NLP library to show a framework to classify documents. That is shown in the middle chapter. I managed to do that but with some missing points: I was not able to improve “tf-idf” or to find a better way to represent documents. I had in mind to do an application about this part but it was totally impossible.

Something I skipped was to develop new kernels for Shogun. I had to choose between developing a new kernel or focusing on the similarity idea, I chose the second one.

Finally, we managed to develop our similarity idea using Shogun's library. As we showed on the last big chapter we had to use a very simple design to be able to finish that in time. We got good results and we are happy to see our prototype working.

The authors from the Shogun's library would like to add what we did and that is the best feedback we can get.

18 They are in the introduction.

6. Further work

The work we are doing in this thesis probably finishes here, since it is not part of a big project no more people will work on it although the DM field is under big development and that means finding better ways to work with heterogeneous data. SVM community is busy with MKL. So we doubt no one will work ever on what we did here but as a developers some new questions or problems arose and although they will get unanswered it is our duty to express them here:

About what we did in the middle chapter (NLP + Shogun to classify documents):

- To implement the framework described in the middle chapter in an fully functional application.
- Work with different document representation.
- Try to use more information from documents, like syntax.

About what we did in the last chapter (similarities + heterogeneous data):

- Explore some interesting side-effects, like how combined features can work with missing values and how to use similarities for more than two training points. This last problem seems very hard because it would require practically to redo what is done about SVM.
- Make similarities return some kind of complex data, not a number. Usually the way a human uses to describe how something is similar is by a description and we know how to calculate the similarity between descriptions (documents). It can be nice to try to adapt this to SVM. It also has to be studied if this is possible.
- Check how using a description (or an image or...) instead of a number as returning value from a similarity condition makes us lose less information.
- Check Mercer's condition to see how a similarity function should be to apply it.
- Check Euclidean Space and Metric Space more deeply to see what is allowed.
- Try this with a real world example not with that “silly toy example”. We talked about some examples like criminal analysis, sickness prediction, robot's movement... We are sure that this will work.
- Create a good “BagOfWords” features not using “StringCharFeatures”.
- Implement all this in a optimal way not using the word “kernel”.

7. Bibliography

[1] C.Cortes and V.N. Vapnik. Support-vector networks. Machine Learning, 20(3):273--297, 1995.

[2] N.Cristianini and J.Shawe-Taylor, An introduction to Support Vector Machines and other kernel-based learning methods, 2000. Cambridge, UK: Cambridge University Press.

[3] M.A.Hearst, Support Vector Machines, 1999. *IEEE Intelligent Systems*, **13**(4), 18–28.

[4] C.Burges, A tutorial on Support Vector Machines for Pattern Recognition, 1998. *Data Mining and Knowledge Discovery*, **2**(2), 121–167.

[5] KR.Müller, S.Mika, G.Rätsch, K.Tsuda and B.Schölkopf, An Introduction to Kernel-Based Learning Algorithms, 2001. IEEE Transactions on Neurla Networks, Vol. 12, NO. 2.

[6] G.Tsoumakas and I.Katakis, Multi-Label Classification: An Overview, 2007. International Journal of Data Warehousing and Mining, 3(3):1-13.

[7] S.Sonnenburg, G.Raetsch, C.Schaefer and B.Schoelkopf, Large Scale Multiple Kernel Learning. Journal of Machine Learning Research,7:1531-1565, July 2006, K.Bennett and E.P.-Hernandez Editors.

[8] S.Bird, E.Klein and E.Loper, Natural Language Processing with Python - Analyzing Text with the Natural Language Toolkit, 2009. O'Reilly Media.

[9] T.Joachims, Text Categorization with Support Vector Machines: Learning with Many Relevant Features, 1997. Springer.

[10] T.Joachims. Making large-scale SVM learning practical. In B.Schoelkopf, C.J.C. Burges, and A.J. Smola, editors, Advances in Kernel Methods - Support Vector Learning, pages 169--184, Cambridge, MA, 1999. MIT Press.

[11] C.-C. Chang and C.-J. Lin, LIBSVM : a library for support vector machines, 2001. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm

[12] Y.Yang and J.O.Pedersen, A Comparative Study on Feature Selection in Text Categorization, 1997. Proceedings of the Fourteenth International Conference on Machine Learning, pages 412-420, ISBN:1-55860-486-3.

[13] P.Soucy and G.W.Mineau, Beyond TFIDF Weighting for Text Categorization in the Vector Space Model, 2005. International Joint Conference On Artificial Intelligence, Proceedings of the 19th international joint conference on Artificial intelligence, Edinburgh, Scotland, pages: 1130-1135.

[14] LL.Belanche, Heterogeneous Neural Networks: Theory and Applications, 2000. Tesi doctoral de Universitat Politecnica de Catalunya, Departament de Llenguatges i Sistemes Informatics.

[15] J.Turmo and H.Rodriguez, Learning rules for information extraction, 2002. Natural Language Engineering 1, Cambridge University Press.

[16] R.Gonzalez, Python para todos, 2009. CC 2.5

[17] B.Eckel, Thinking in C++, 2000. 2nd ed. Volume 1. http://www.smart2help.com/e-books/ticpp-2nd-ed-vol-one/Frames.html.

8. Our contribution

Our contribution to the DM world is as we showed a new way to work with heterogeneous data. We explained it theoretically and later we proposed a way to implement it in a working library.

Our contribution can be summarized as follows: When dealing with heterogeneous data create (or use) a similarity function for each type on input (or for each input) and then create a similarity that will combine them all. Provide that functions to the SVM algorithm and it will work.

Similarities are easy to construct because what we need is human knowledge and also because we can make them to be a powerful tool. Similarities are more powerful than kernels because they are not restricted to the inner product.

We would like to show the real power of this with a real world example dealing with really different kind of inputs.

A. Appendix – Sample test script

What follows is an example of script as the ones we used in the middle chapter to test the results and different techniques. Comments are inside the code. No lines are removed and if pasted that code would run and show the results in the console. As any Python code, it can be run using the Python's interpret pasting the code in the interpret or saving the code in a file and running "python name_of_file". The last line should be properly indented.

```python
import nltk
from nltk.corpus import movie_reviews
from nltk.corpus import stopwords
from nltk.corpus import wordnet as wn
import os

from numpy import *
from numpy.random import randn
from shogun.Features import *
from shogun.Classifier import *
from shogun.Kernel import *

# given a word we calculate its idf value that we will use in the next function
def calculate_idf(word):
  InDoc = 0
  InDoc = sum(word in i for i in documents_as_set)
  return math.log(float(1500./(1+InDoc)))

# for each document returns its tfidf representation
def document_features_tfidf(document):
  '''the document must be a list, not a set'''
  document_words = set(document)
  words_freqs = nltk.FreqDist(document)
  features = []
  for word in word_features:
    if (word in document_words):
      # tf x idf
      features.append(words_freqs.freq(word) * total_idf[word])
    else:
      features.append(0.0)
  return features/sqrt(sum(n*n for n in features))


# the bell's width, a quick way to calculate it
def calculate_gauss_width(train_set):
  '''ugly and quick way to get a Gaussian width'''
  ones = 0
  for i in range(len(train_set)):
    newones = sum(1 for feature in train_set[i] if feature > 0)
    ones = ones + newones
  return ones/len(train_set)


stopWordsEng = set(stopwords.words('english'))
m = {'pos': 1., 'neg': -1.}

# This is a powerful example of list comprehesion in Python
# we are removing non-alphanumeric words and stopwords
# we are also lemmatizing all the words with morphy
# and for each document we create a tuple with the words and the label
documents = [ ([wn.morphy(word) for word in movie_reviews.words(fileid)
          if word.isalpha() and word.lower() not in stopWordsEng]
          , m[category])
          for category in movie_reviews.categories()
          for fileid in movie_reviews.fileids(category) ]

# if a word does not exist morphy returns "None", we need to remove
# all the "None" entries
documents = [ ([word for word in doc if word is not None],c)
              for (doc,c) in documents ]

random.shuffle(documents)

# working with a set makes calculate_idf faster, it was very very slow
documents_as_set = [ set(words) for (words, label) in documents]
categories = [c for (d,c) in documents]

# this for is making the 4-fold cross validation
for x in range(0,2000,500):
  print "CROSS VALIDATION",x/500.
  start, end = x, x+500

  # we separate the labels between train and test
  trainlab = categories[:start] + categories[end:]
  testlab = categories[start:end]

  # for all the documents in the train set, we want to know the frequency
  # of their words
  all_words = nltk.FreqDist(word for (words,labels) in documents[:start] + documents[end:] for word in words)
  # we choose the top 2000 words (the 2000 words more frequent)
  word_features = set(all_words.keys()[:2000])
  # for that top words we calculate its idf value
```

```
total_idf = dict((w, calculate_idf(w)) for w in word_features)

# we extract the features from each document, each document will be
# represented by the tf-idf of its words
featuresets_tfidf = [document_features_tfidf(d) for (d,c) in documents]
# we separate the features for training and test
train_set_tfidf = featuresets_tfidf[:start] + featuresets_tfidf[end:]
test_set_tfidf = featuresets_tfidf[start:end]
feats_train_tfidf = RealFeatures(array(train_set_tfidf).T)
feats_test_tfidf = RealFeatures(array(test_set_tfidf).T)

labels = Labels(trainlab)

width = calculate_gauss_width(test_set_tfidf)

###################################
#REAL EXAMPLES FROM ROTTEN TOMATOES
rotten_tomatoes = []
for file in os.listdir("/home/juanma/rotten_tomatoes/neg"):
  f = open("/home/juanma/rotten_tomatoes/neg/"+file, "r")
  raw = f.read()
  tokens = nltk.word_tokenize(raw)
  text = nltk.Text(tokens)
  rotten_tomatoes.append((text,-1.))

for file in os.listdir("/home/juanma/rotten_tomatoes/pos"):
  f = open("/home/juanma/rotten_tomatoes/pos/"+file, "r")
  raw = f.read()
  tokens = nltk.word_tokenize(raw)
  rotten_tomatoes.append((tokens,1.))

random.shuffle(rotten_tomatoes)

rotten_formatted = [([wn.morphy(word) for word in review
            if word.isalpha() and word.lower() not in stopWordsEng] , c)
            for (review,c) in rotten_tomatoes
            ]

rotten_formatted = [ ([word for word in doc if word is not None],c)
              for (doc,c) in rotten_formatted ]

featureRotten = [document_features_tfidf(d) for (d,c) in rotten_formatted]
rotten_test = RealFeatures(array(featureRotten).T)
```

```
categoriesRotten = [c for (d,c) in rotten_formatted]
labelsRotten = Labels(array(categoriesRotten))
# END OF REAL EXAMPLES FROM ROTTEN TOMATOES
###########################################

print "STARTING TEST....WIDTH: ", width

# i will be used for Capacity
for i in (0.5, 1., 10., 100.):
  #width goes from 1.5*width to 5*width
  for j in (1.5, 2., 5.,):
    actual_width = width * j
    # we create the kernel with the train features and the chosen width
    kernel = GaussianKernel(feats_train_tfidf, feats_train_tfidf, actual_width)
    # this kerenel is used to create the svm object
    svm = LibSVM(i, kernel, labels)
    # the svm is trained, it finds the support vectors
    res = svm.train()
    # we check it against our test set
    out = svm.classify(feats_test_tfidf).get_labels()
    acc = sum(sign(out)==testlab)
    print "C: ",i, "W: ", j, " ",round(acc/500.,2),
    # we also check it against the documents from rotten tomatoes website
    outRotten = svm.classify(rotten_test).get_labels()
    acc = sum(sign(outRotten)==categoriesRotten)
print round(acc/60.,2)
```