Universitat Oberta de Catalunya

Artificial Intelligence - CET1

# Dimensionality reduction and Clasification

**Juan José Rodríguez Aldavero**

April 28, 2020

# Contents

# 1 Exercises

## 1.a Principal Component Analysis

In this section we will perform a dimensionality reduction procedure using the **Principal Component Analysis (PCA)** method. To do this, we will create a script in Python using the Scikit-Learn library.

The PCA method is a procedure used to reduce the dimensionality of a dataset, while maintaining as much as possible the original structure of the data. To this end, the dataset is projected on a subspace such that the loss of information associated with this projection is minimized. This is equivalent to looking for a projection whose variables have maximum variance (since the variance of each variable can be understood as the amount of information it stores). In this way, it can be mathematically proven that the projection whose variables have maximum variance will be on the subspace associated with the eigenvectors of the covariance matrix.

Thus, the procedure is very simple: given a data matrix $\hat{X}(n \times p)$, first the algorithm calculates the $S$ matrix of variances and covariances, then all the $p$ eigenvectors of the matrix are found and then the original data is projected on the subspace generated by the first $r$ vectors, where $r$ is the final dimension we want for the data. The eigenvectors are called **principal directions**, and the eigenvalues are called **principal components**. Clearly, the number of principal directions, $p$, will match the number of variables in the original problem.

We proceed to generate the script using the scikit-learn library.

A. **How many principal components are required to represent 95% of the variance of the original data?**
   To calculate the number of principal components that represent 95% of the total variance, we created a script that calculates the variance of each principal component, and then sum them until we reach the desired threshold. The result obtained is **23 variables**, reaching a cumulative variance of 95.7%.

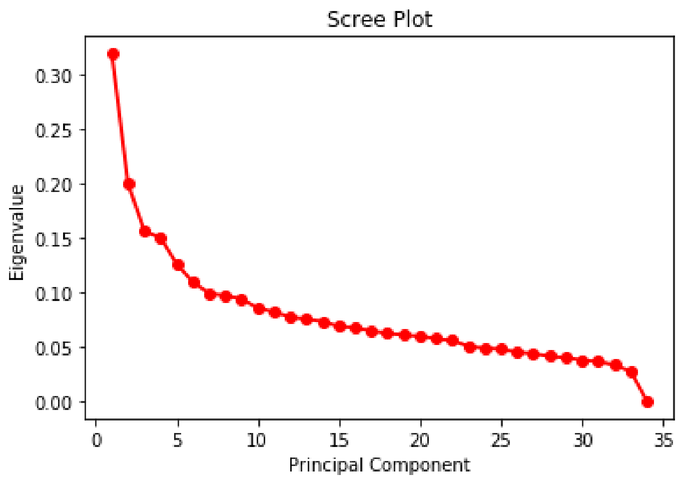

*Figure 1.1: View of the 34 principal components (eigenvalues of S)*

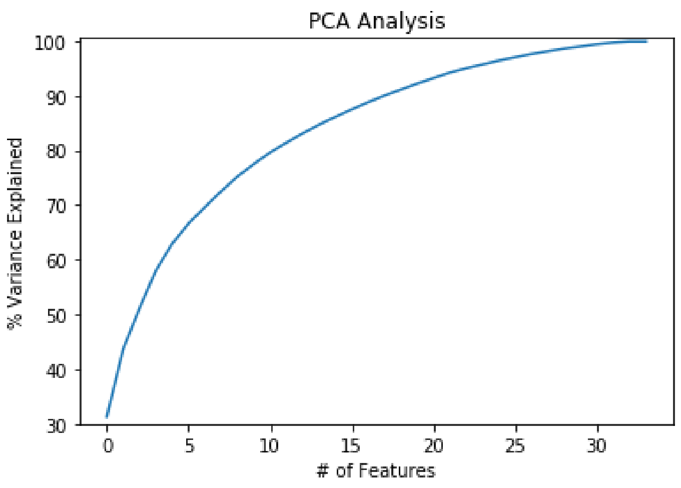

*Figure 1.2: Cumulative variance explained by the principal components*

B. **Rebuild the data set from the 18 components and calculate the loss of information with respect to the original set.**
In this section we calculate the PCA projection for the first 18 components and then rebuild the initial set by performing the reverse transformation. This can be done since sklearn has the *inverse_transform()* method that calculates it. Applying this script, we obtain values similar to the original dataset, and a **total loss of information of 2.7%**, where we have defined the loss of information as:

$$loss(\hat{X}) = mean(\hat{X}_{orig}^2 - \hat{X}_{rec}^2)$$

C. **Display the original data and the transformed data according to its two main components**
We calculated the graph composed of the first two original variables (blue) and the first two principal components (red).

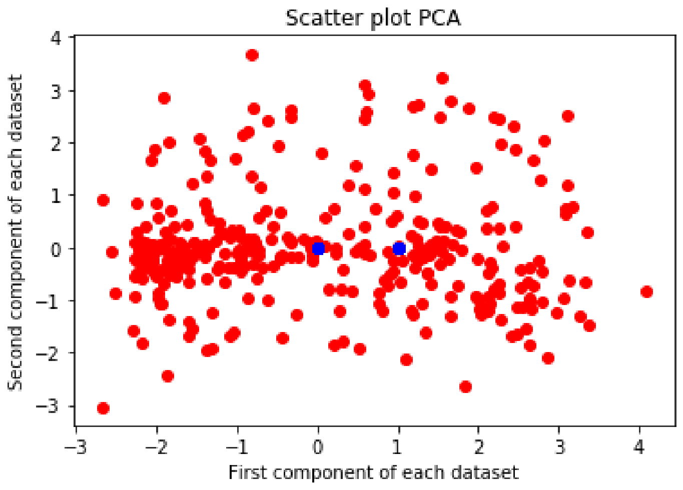

*Figure 1.3: PCA*

We see how the first two original variables do not show much information since they are only pairs of ones and zeros. On the other hand, the points associated with the first two main components present a **high degree of variance**. This was expected because as it was explained, the new principal components are looked such as they have maximum variance. Also, the linear combinations of the original variables that give rise to the main components may contain information about the problem. From the first variable, conclusions can be drawn about "size factors", while from the rest, "shape factors" can be obtained.

## 1.b Multidimensional Scaling

In this section we will perform a **Multidimensional Scaling (MDS)** procedure to the problem dataset. This procedure is analogous to PCA and consists in calculating a set of $p$ variables $\{y_1, \ldots, y_p\}$ such as to justify the distances stored in the similarity matrix. In other words, it is a technique that allows the calculation of a map showing the relative distances between a set of elements, knowing only the distances between them, and without the need to have any knowledge of the underlying data set.

If the similarity matrix comes directly from a data matrix $\hat{X}(n \times p)$, we will say that we are doing **metric scaling**, since these distances correspond to Euclidean metrics. However, if it does not come from a data matrix but is synthetically constructed then we say we are doing **non-metric scaling**. An example of non-metric scaling is the construction of the similarity matrix by means of surveys or "expert knowledge" and has applications in many fields such as marketing or sociology.

Mathematically, the method consists of calculating the first $p$ eigenvectors of the synthetic or associated data matrix

$$\hat{Q} = \frac{1}{n}\hat{X} \cdot \hat{X}$$

and project the original data set onto the first $r$ eigenvectors (usually $r = 2$).

We calculated the method for the original data set three times in a row and observed the resulting graph by painting a different color for each run. The result is as follows:

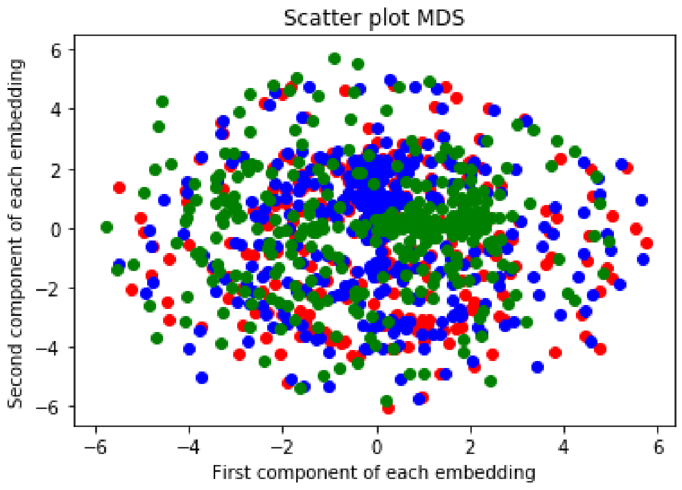

***Figure 1.4:*** *MDS representation for three consecutive executions*

We see how each execution results in a different outcome. This is because **there is no single solution for MDS**. In other words, given a set of distances between elements, there is no unique distribution of these elements that gives rise to the distances. For this reason, sklearn's MDS algorithm incorporates a random parameter that allows exploring the solution space for each execution, which may be useful when looking for new applications for the method.

Finally, MDS is useful because it allows interpretation of a data set. Given a data set $\hat{X}(n \times p)$ it is always possible to find a matrix of similarities $Q$ and from there obtain a two-dimensional map applying MDS. This map will hopefully contain information on the original variables in the form of clusters or structures and will allow a better understanding of the nature of the problem.

## 1.c Classification algorithms

In this section we will apply five classification algorithms from the scikit-learn library on the dataset to obtain the parameters *score*, *training time* and *prediction time* using cross-validation. In particular, we will apply the following algorithms:

A. **k-Nearest-Neighbors**
The principle behind nearest neighbor methods is to find a predefined number of training samples closest in distance to the new point, and predict the label from these. The number of samples can be a user-defined constant (k-nearest neighbor learning), or vary based on the local density of points (radius-based neighbor learning). The distance can, in general, be any metric measure but standard Euclidean distance is the most common choice. It is implemented on sklearn as the class *neighbors.KNeighborsClassifier()*.

B. **Linear SVM**
An SVM model is a representation of the examples as points in space, mapped through a mathematical function named kernel so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on the side of the gap on which they fall. The linear version (linear kernel) is implemented on sklearn as the class *svm.LinearSVC()*.

C. **Decision Tree**
a Decision Tree is a non-parametric supervised learning method used for classification and

regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. It is implemented on sklearn as the class *tree.DecisionTreeClassifier()*.

D. **AdaBoost**
An AdaBoost classifier is a meta-estimator that begins by fitting a classifier on the original dataset and then fits additional copies of the classifier on the same dataset but where the weights of incorrectly classified instances are adjusted such that subsequent classifiers focus more on difficult cases. It is implemented on sklearn as the class *ensemble.AdaBoostClassifier()*.

E. **Gaussian Naive Bayes**
Naive Bayes methods are a set of supervised learning algorithms based on applying Bayes' theorem with the "naive" assumption of conditional independence between every pair of features given the value of the class variable. In Gaussian Naive Bayes the likelihood of the features is assumed to be Gaussian:

$$P\left(x_i|y\right) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{\left(x_i - \mu_y\right)^2}{2\sigma_y^2}\right)$$

It is implemented on sklearn as the class *naive_bayes.GaussianNB()*.

Once we have seen the description of each method, we proceed to create a script that calculates the parameters *score*, *training time* and *prediction time*. To do this, we use the cross-validation method of the *model-selection* class to perform the cross-validation and then we calculate the values of these variables for five folds. Then, we calculate the mean and the statistical deviation of this set of values in order to interpret them more clearly. The results are contained in the following table, where for simplicity we have noted the standard error $SEM = \frac{\sigma}{\sqrt{n}}$ for two significant figures.

| | **Score** | **Training time (ms)** | **Prediction time (ms)** |
|---|---|---|---|
| **kNN, 3** | $(83,19 \pm 1,42)\%$ | $0,999546 \pm 0,000072$ | $5,996418 \pm 0,000071$ |
| **kNN, 4** | $(84,33 \pm 1,23)\%$ | $0,99983 \pm 0,00014$ | $6,196 \pm 0,089$ |
| **kNN, 5** | $(82,62 \pm 0,87)\%$ | $0,99959 \pm 0,00018$ | $5,796 \pm 0,089$ |
| **LinearSVC** | $(84,34 \pm 1,35)\%$ | $2,199 \pm 0,089$ | $0,200 \pm 0,089$ |
| **DecissionTree** | $(87,20 \pm 0,95)\%$ | $4,199 \pm 0,089$ | $0,99864 \pm 0,00012$ |
| **AdaBoost** | $(91,47 \pm 1,09)\%$ | $147,91 \pm 4,45$ | $9,60 \pm 2,71$ |
| **Gaussian Naive Bayes** | $(86,60 \pm 0,63\%)$ | $0,999355 \pm 0,000093$ | $0,60 \pm 0,11$ |

We see how all the results are quite similar in score, not so much in execution time. To see it more clearly, the following graphs have been made:

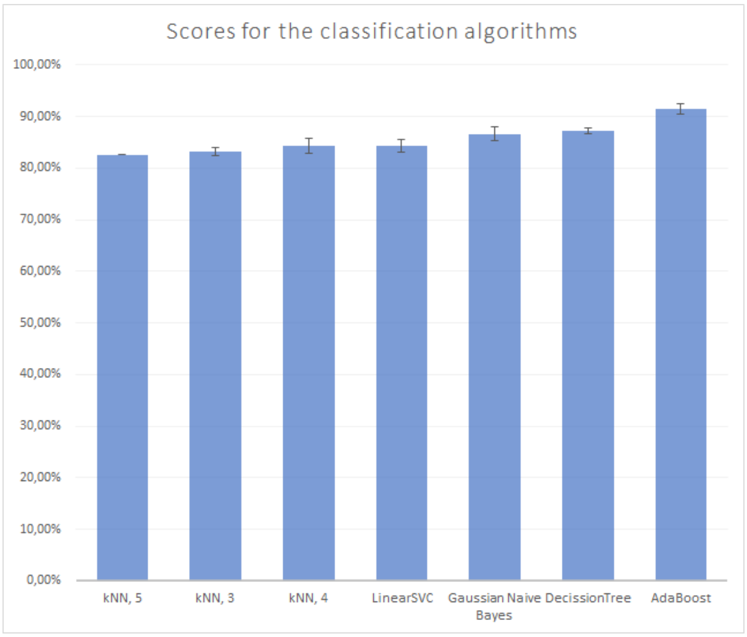

***Figure 1.5:*** *SVM scores for diferent kernels and C values*

We see how AdaBoost performs the best, with an average performance over 90%. The worst performing algorithms are the kNN, with the interesting property that the performance drops as the number of neighbours increases (maybe because of overfitting).

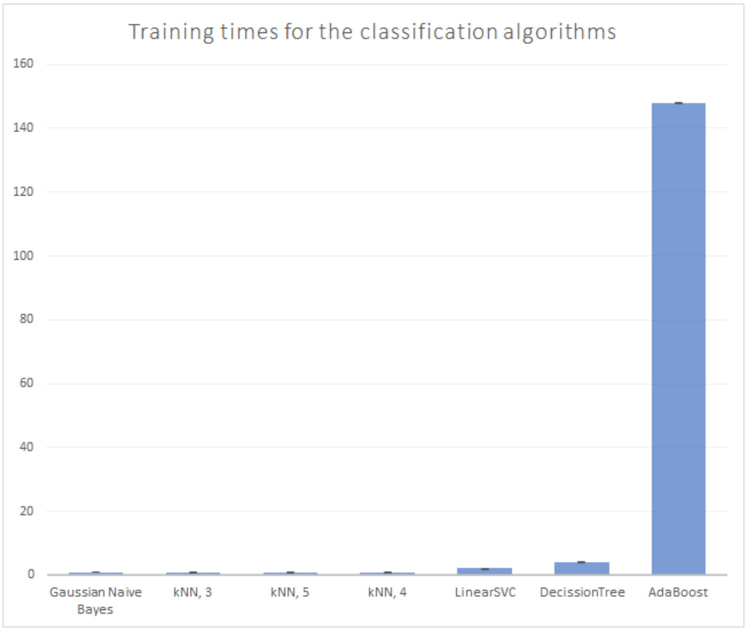

***Figure 1.6:*** *SVM scores for diferent kernels and C values*

We see how AdaBoost's training time completely dominates the rest, being almost 40 times longer than DecissionTree. The fastest method is Gaussian Naive Bayes.

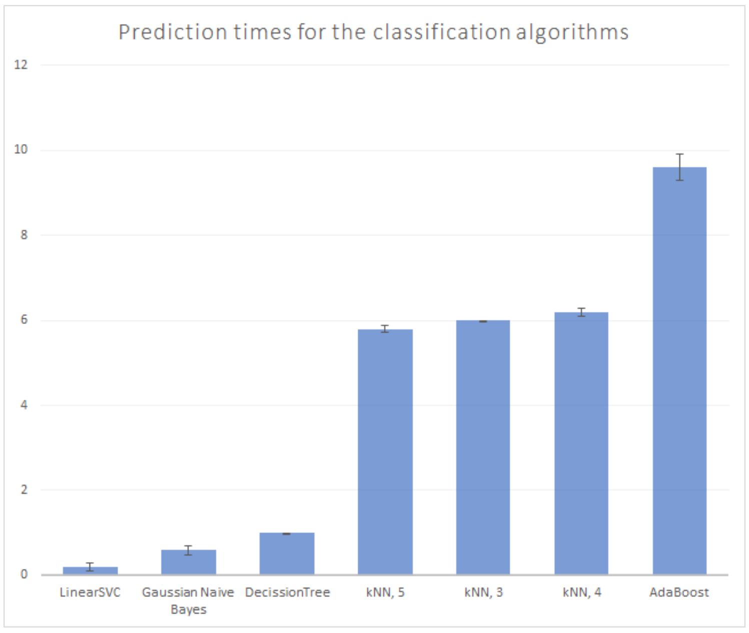

***Figure** 1.7: SVM scores for diferent kernels and C values*

We see that the prediction times are not as different as the training times, but AdaBoost is still the slowest, followed by kNN. From this result, it seems that AdaBoost can be suitable for results that need higher quality even at the cost of more computational resources, while other methods such as GNB can be useful when you want faster results at the cost of prediction accuracy.

## 1.d Support Vector Machine

In this section, we finish the deliverable by studying the **Support Vector Machine (SVM)** classifier in the library for four possible kernel configurations.

The SVM method is an algorithm used for classification and regression problems based on finding the set of hyperplanes that best divides the dataset into the target classes. To find these hyperplanes it relies on the so-called support vectors, which are the points closest to the hyperplanes and which, if removed, would modify the position of these hyperplanes. The procedure by which these planes are found is called **kerneling**. Kerneling is a method of dimensionality expansion by which a non-linear mathematical function $\phi(x)$ is applied to the data and corresponding hyperplanes are found on the image of this function so that the decision surfaces on the original data set are non-linear.

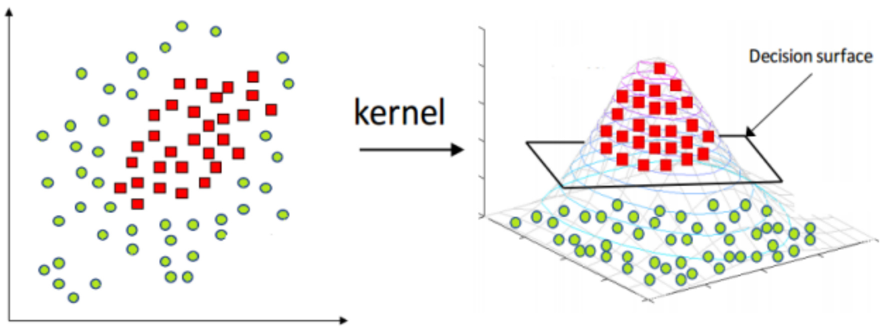

*Figure 1.8: Example of SVM kernelling*

We see how the hyper plane of separation in the image on the right corresponds to a non-linear decision boundary in the image on the left.

In particular, we calculate values for the following four SVM configurations:

A. Linear kernel, $C = 0.025$
B. Linear kernel, $C = 100$
C. RBF kernel, $C = 0.025$
D. RVF kernel, $C = 100$

In the following table we see the results:

| SVM | Score | Training time (ms) | Prediction time (ms) |
|---|---|---|---|
| **Linear, 100** | (88,33 ± 0,90)% | 79,35 ± 13,30 | 0,999641 ± 0,000051 |
| **Linear, 0.025** | (84,91 ± 1,08) % | 3,60 ± 0,18 | 0,999928 ± 0,000067 |
| **rbf, 100** | (92,59 ± 0,89)% | 4,197 ± 0,089 | 1,00021 ± 0,00019 |
| **rbf, 0.025** | (64,105 ± 0,081)% | 5,996656 ± 0,000021 | 1,60 ± 0,11 |

In order to have more clarity in interpretation, we make the following graphs of the data:

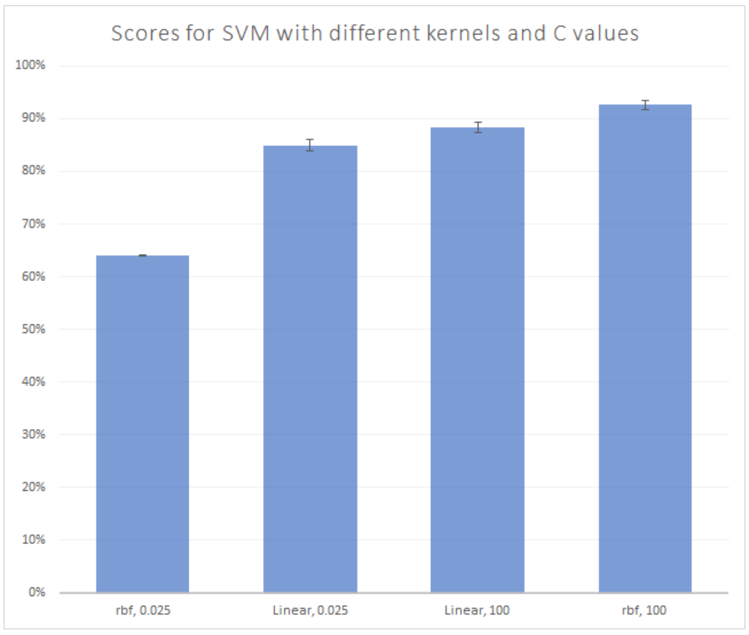

***Figure 1.9:*** *SVM scores for diferent kernels and C values*

We see that the method that achieves the highest accuracy is the rbf with $C = 100$, followed by the linear with this same value of $C$. Therefore, it seems that a higher value of this parameter increases the accuracy of the model.

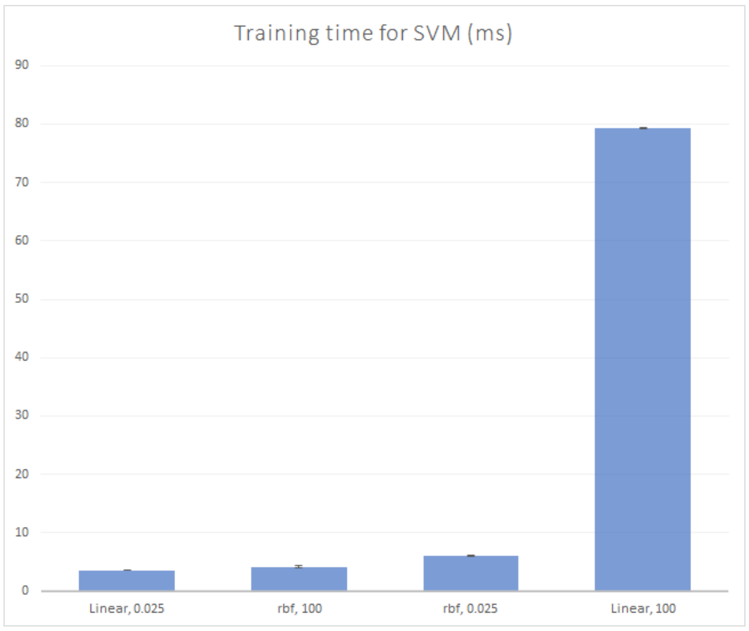

***Figure 1.10:*** *SVM training times for diferent kernels and C values*

We see that increasing the value of $C$ considerably increases the training time for the linear model, not so much for the rbf model. This can be a considerable advantage of this kernel, since it presents a high degree of accuracy without sacrificing performance as it was previously the case with AdaBoost.

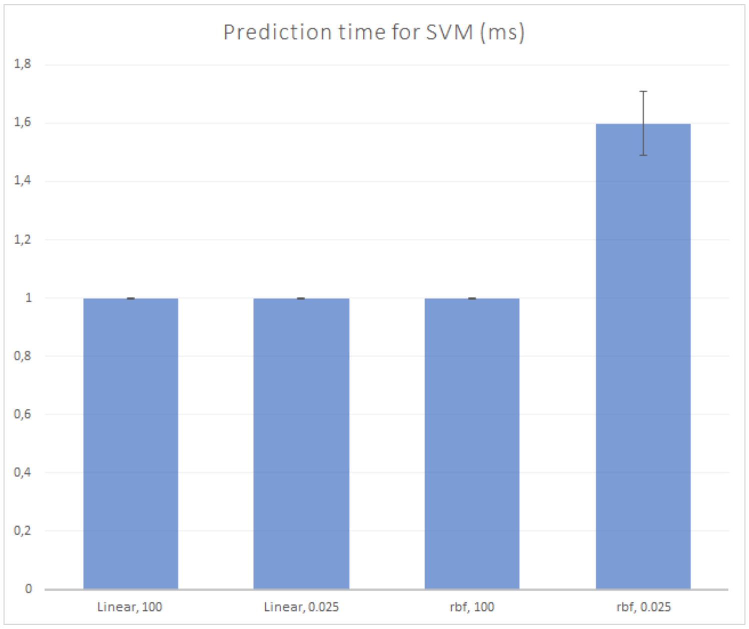

***Figure 1.11:*** *SVM execution for diferent kernels and C values*

Finally, we see how the prediction times remain constant around the millisecond except for the rbf model with $C = 0.25$ which strangely separates from the rest and presents a relatively high degree of uncertainty.

Having seen the results, it is useful to know a little about the technical background of the models. SVM algorithms use a set of mathematical functions that are defined as the kernel. The function of kernel is to take data as input and transform it into the required form. Different SVM algorithms use different types of kernel functions. These functions can be different types. For example linear, nonlinear, polynomial, radial basis function (RBF), and sigmoid. In the particular case of sklearn, we can use the above kernels in addition to a kernel type called "precomputed".

The C parameter tells the SVM optimization how much you want to avoid misclassifying each training example. For large values of C, the optimization will choose a smaller-margin hyperplane if that hyperplane does a better job of getting all the training points classified correctly. Conversely, a very small value of C will cause the optimizer to look for a larger-margin separating hyperplane, even if that hyperplane misclassifies more points.

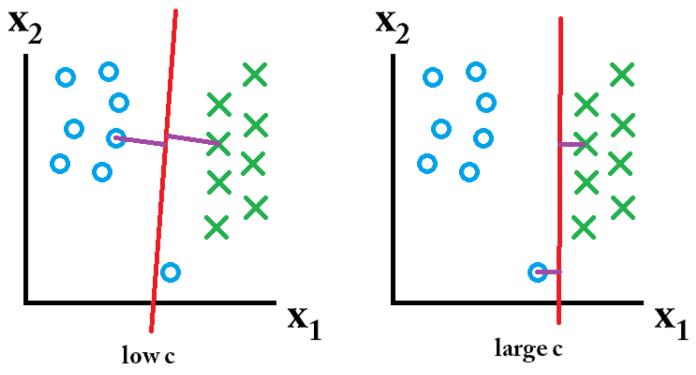

***Figure 1.12:*** *Diagram of the effects of the C parameter. Observe how for larger values of C the gap is narrower*

We see how increasing the value of C and demanding good hyperplanes to the model gives good results for this data set, perhaps because the structure of the data is well adapted to an SVM. This parameter can only take positive values, even if they are arbitrarily large or small.