LINMA2472 - Algorithms in data science

Project 2: k-PCA and random projections

Authors : Heroufosse Gauthier
Degives Nicolas
Gandjeto Martine

2021 - 2022

Introduction

In this second project 2, we got two different objectives :
The first is to implement an outlier detection using different types of algorithm such as PCA and kPCA. We will then compare their respective performance in order to chose the better one for our problem.
The second part of this project is to compare the fastness of different algorithm in task resolution. We will use a linear and a Gaussian kernel SVM, and a random Fourier features (RFF). After comparing the testing time, we will make vary a parameter in order to see the influence it has in both time performance and accuracy.

Assignment 1 : PCA vs. k-PCA for outliers detection

In this second project 2, the goal is to work on embeddings. An embedding can be defined as the mapping of one set into another. In other words, an embedding tries to represent some mathematical structure in a clearer way than the initial one. In this assignment, we will compare two well-known embeddings, PCA and k-PCA, in outlier detection. An outlier is defined as a point that is sampled from another distribution than the one we consider.

The data consists in two arrays $X \in R^{N\times 2}$ and $\mathbf{y} \in R^N$ with :

$$y_i = \begin{cases} 1 & \text{if } \mathbf{X}_{i,*} \text{ is an outlier,} \\ 0 & \text{if } \mathbf{X}_{i,*} \text{ is a true point.} \end{cases}$$

with N the number of samples. There are N_o outliers and N_t true points, that is N_t points that are not outliers.
After performing a bit of EDA, we will split our data into a training and a testing set in order to analyze the performance of both models. Based on that, we will then selection the best fitting model for outlier detection and evaluate the generalization error of that classifier.

1.1: Performing EDA on our data set

In order to get an initial overview of our data, we perform an exploratory data analysis. First, basic statistics such as mean, median and variance of the data were calculated. We also compute the outlier ratio r_o, and found that it is equal to 0.2 which means that 20 % of the data are outliers. As this ratio is quite high, we cannot confidently analyze those basic statistics, because they could be highly influenced by the outliers.
To get more information about those seditious points, we plotted the true data and the outliers on the same figure, as you can see in the Fig 1. The data structure is clearly composed by four area that forms a rectangle together. Outliers are randomly arranged around the true data points.

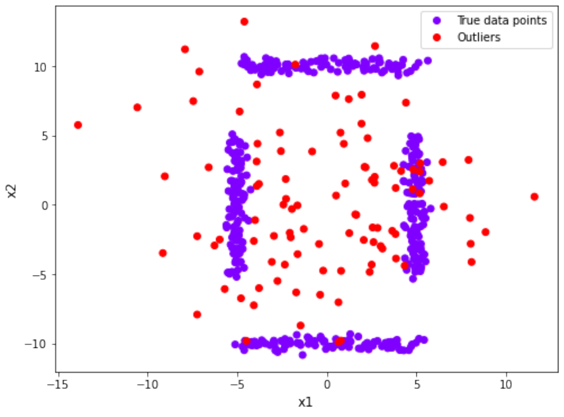

Figure 1: Visualization of the dataset separating true data and outliers

Due to this particular structure, it is not possible to well represent the data on one axis (which will probably be parallel to the x1 axis). Therefore, detecting outliers using PCA will not be easy. We consider as succes metric the number of positive hits, that is :

$$score = \frac{TP + TN}{P + N}$$

.

The minimum score that is acceptable for any outlier detector is more than 0.5. In fact, if the minimal score is 0.5, it means that we have at least even chances to get the same result by splitting randomly the outlier sample.

1.2: Splitting of the main data set

The next step was to split the dataset in order to then train the models. We used the already implement `train_test_split` function that we imported from the `sklearn` python package. We split the dataset into three parts : the training, validation and test sets. Training and validation sets are available for parameters fitting, and the test set will measure the accuracy of the model at the end. Of course, our algorithms will not train with this test set, or they will already know where the data should be classified. We used a 90/10 repartition of the data between train/validation side and the test side. Then, we assign 70 % of the training set for training and 30 % for validation.

It was necessary to shuffle the data when splitting because the data set was initially ordered. All the outliers are at the end of the data set. To deal with that, we set the parameter `shuffle` to `True` in the `train_test_split` function.

1.3: Outlier detection with PCA

Now that the data is ready, we can perform the outlier detection. We will go first with "normal" PCA and then talk about k-PCA in the next section.
We imported an abstract class from `sklearn` to build our classifier `OutlierDetectorPCA`. After updating a few lines in the given code, we fill some holes in the instructions of some of these functions.Then, we test our implementation by computing the validation score for k = 1. The score that we obtain is around 0.83.

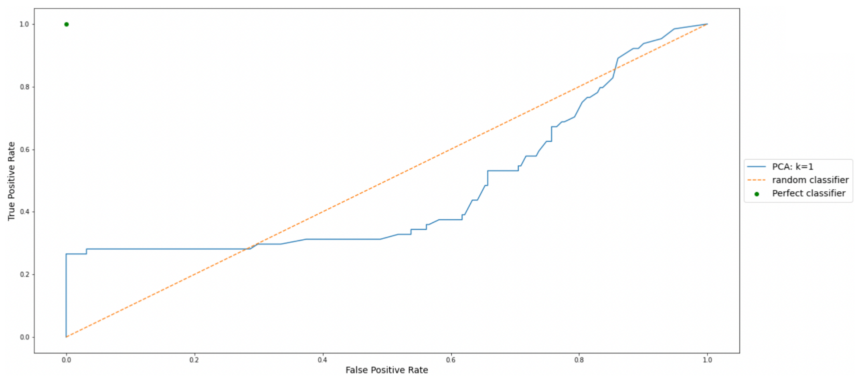

Figure 2: ROC curve from the implementation of the PCA outlier detector for k = 1

We identified a few parameters that influence the model's behaviour. First, we got k. It represents the size of the subspace we use to predict if the data is an outlier or not. By default when talking about PCA, k should be equal to 1.
The second parameter we identified is τ. It represents the threshold between outlier and non-outlier value. To truly understand it, we need to go a little bittle more in detail here. As an observation is evaluated by our algorithm, it returns a reconstruction error, that is the euclidean distance measured between the observation and her projection on the PCA subspace. τ define the acceptance threshold for a given reconstruction error. The larger it is, the more conciliatory it is.

1.4: Outlier detection with k-PCA

- Explain what is done in fit:
  For simplicity, the fit function performs the kernel-PCA in two steps. First, a high-dimensional features data are explicitly computed. Basically a kernel function (either the radial basis function or the polynomial function) is used to map our initial features into a high-dimensional space. For instance using our training data that contains 2 features and 315 samples we get a new matrix with 314 features. Second, a standard PCA is performed using the new high-dimensional feature matrix.

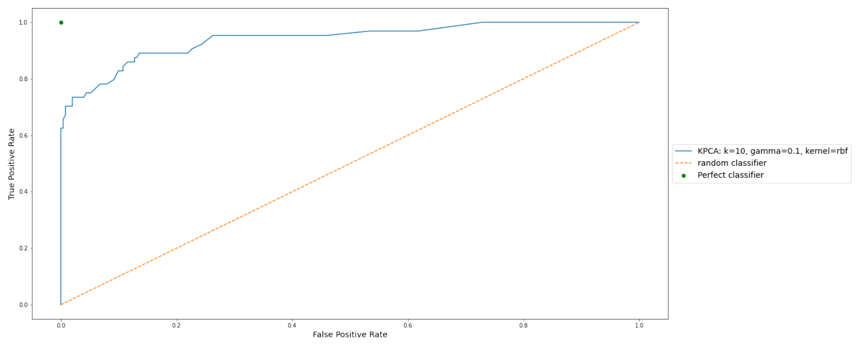

Figure 3: ROC curve from the implementation of the k-PCA outlier detector for k = 10,$\gamma = 0.1$ and rbf kernel

- Discussing the value of k:
  k is the number of eigenvectors to keep. In the standard PCA that we have previously performed, the value of k was between 1 and 2 (since we have only 2 features). In the case of the kernel-PCA we are using a high-dimensional feature matrix. Therefore the value of k can vary in a wider range. For instance using the training set (with 314 features after applying the kernel) the value of k could vary between 1 and 314.
- The hyperparameters of the kPCA are k, gamma and the type of kernel, because they are defined by us and not derived by the model.
- The parameters are similar to the PCA model, we have τ and also the eigenvectors.

1.5: Model selection

Following the shape of the data, we already knew that k-PCA would give better results than PCA. We instanced two nested loops we arbitrary chosen values of k and γ, and then we choose the best score for the combination of two values. By testing some combination of the parameters k and γ, we found that the optimal values are respectively **10** for k and **0.001** for γ (arronded). Those values return the best score for outliers classification, **0.963**.

1.6: Analysis

We have used our best model which is the kpca (k=10, $\gamma = 0.001$, kernel='rbf') to assess the performance on the test set. The table below displays the score obtained. It can be seen that the score using the test set is slightly lower than for the training and the validation sets. This seems to indicate that our model does not generalize very well.

Set	Score
Training	0.95
Validation	0.98
Test	0.92

Table 1: Score of the kPCA model using different sets of data

Assignment 2 : Random Fourier features (RFF)

In this part, we use and train three different classifiers to demonstrate the speed gains by using RFFs.
Those three classifier are :

- A linear SVM
- A kernel SVM (with Gaussian kernel)
- Another linear SVM on the data with RFF

For this purpose, we've used the "MNIST" data set which we are going to split into 10.000 training instances and 60.000 testing instances.

2.1: Testing time

The figure 4 compare the time taken for each method to classify the 60.000 testing instances (this does not include the training time).

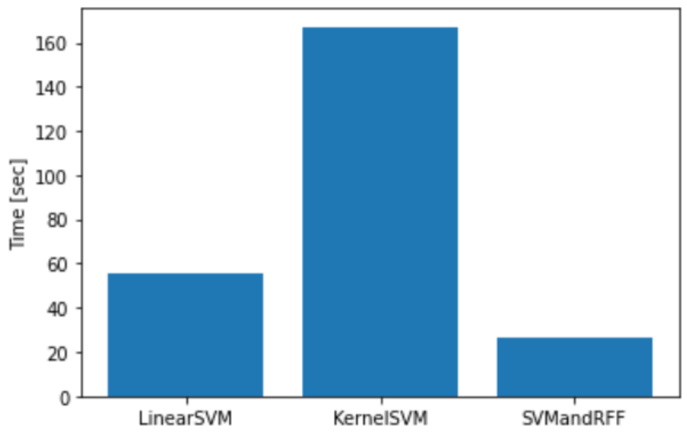

Figure 4: Time taken to classify the test instances

The Kernel SVM method take clearly more time than the linear SVM one. This make sense knowing that linear SVM is the most basic type and is usually one dimensional. Kernels are used for solving nonlinear problems by transforming the feature space which take times to compute but can suit to nonlinear problems. The third method is also a linear SVM but we first have applied RFFs which seems to reduce the computation time. This is because RFFs simplifies the datas which are then more easily usable for the linear SVD However, in this case the score of each method are very similar so it would be interesting to use the fastest method.

2.2: Training time and parameters

In order to see the influence of the size of D, we run the third model mentioned above. The initial value of D was 300 so we are going to try with D ϵ [200,250,300,350,400] to see how it change around 300.

We can see on the figure 5 that the accuracy of linear SVM on RFFs data increase whit D but this precision require more computation time.

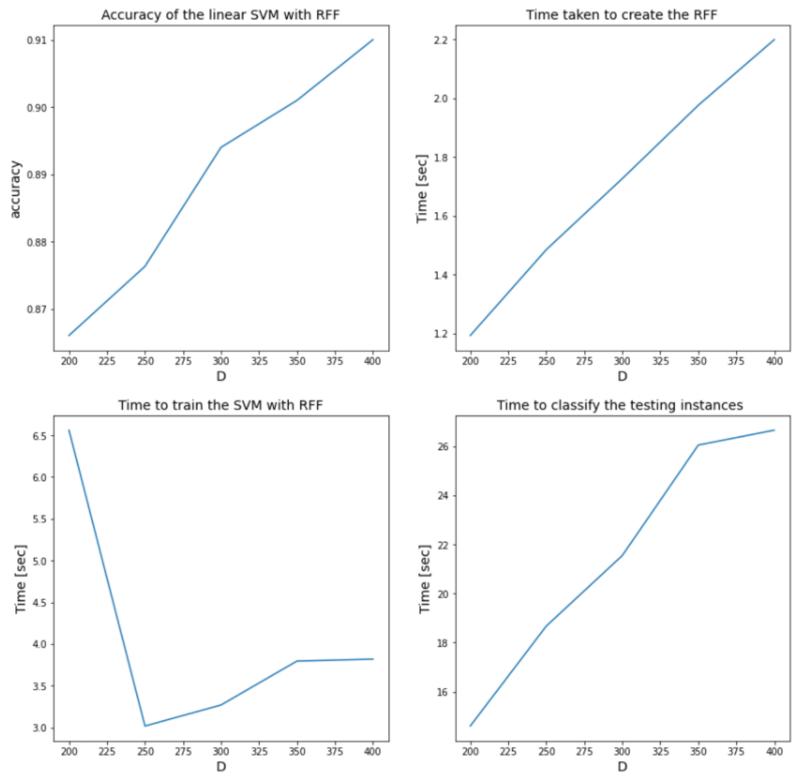

Figure 5: Variation of computation time and accuracy of SVM on RFFs data