Project 2 - Data Processing

Data Mining

Óscar Cubeles Ollé

December 2022

1 Project Goal

The main objective of this data mining project is to apply the knowledge of Instance Based Learning and preprocessing through the use of algorithms such as PCA to a real project using the Python programming language. In this way we will learn how to implement Instance based learning algorithms, how to do selection, preprocessing and attribute search.

For this, you will use different Python libraries like scikit-learn which is oriented to data analysis, NumPy to execute mathematical operations and matplotlib to make plots and show data.

2 Theoretical Context

Before starting with the case study and the implementation, it is necessary to make a previous introduction to the concepts that we are going to apply in this project in order to ease the implementation comprehension.

2.1 What is PCA and SVD?

In this project, one of the algorithms used for preprocessing is Principal Component Analysis, which in short is PCA. This algorithm is classified as a signal representation algorithm, i.e. its objective is to map the actual data into a lower dimensional space. So, it tries to reduce the number of dimensions of a dataset.

The goal is not only to reduce the number of dimensions, but also to preserve randomness without losing important data.

The technique that PCA uses to do the dimension reduction is feature extraction, which is based on the idea that by using the features of the original data, it creates a new subset of data features as a combination of the current one that reduces the dimension of the data.

To do this, the PCA calculates principal components, which are new variables that are constructed as linear combinations or mixtures of the initial variables. And these new variables are orthogonal and capture the maximum variance of the data, so that components that have less variance are not used. The following figure shows an example of a PCA.

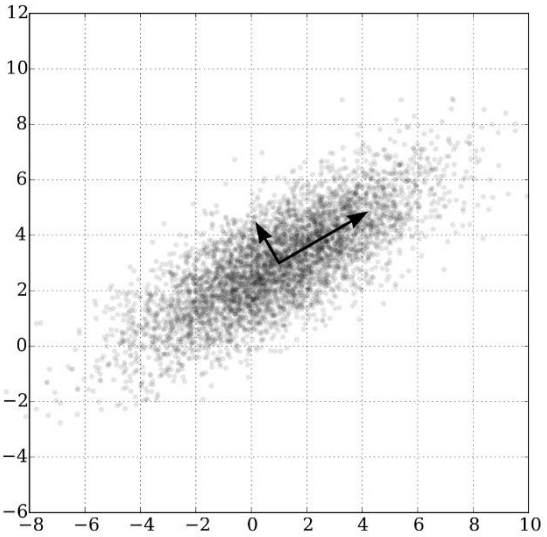

Figure 1: PCA example *(Wikipedia, 2022)*

As can be seen in the image above, two orthogonal components with high variance are obtained from a set of data using PCA.
In this project, the concept of Singular Value Decomposition (SVD) is also used. Unlike PCA, SVD is not a technique to reduce the dimensionality of the data. Rather, it is a technique for decomposing a matrix into three matrices, one of which is the diagonal of eigenvectors, which can be very useful in some respects. The relationship between SVD and PCA is that SVD extracts data in the direction of highest variance, which is useful to compute the principal components in PCA.

2.2 What is a K-NN Classifier?

The K-NN classifier is a type of instance based learning (IBL) algorithm, also called memory based learning. This type of algorithms is also known as lazy learning algorithms since they only perform computations when a new instance arrives. IBL algorithms are based on comparing new instances with known instances already stored in memory to predict the outcome of the new instance.

The K-NN classifier is part of the IBL algorithms and focuses on the idea that all instances are points within an n-dimensional space and defines a neighbour instance in terms of its distance from its nearest instances.

The K in the K-NN algorithm stands for the number of neighbours to look at when a new instance arrives. And to predict the outcome of this instance,

what is done is to look at the k nearest instances and classify it in the same way as most of the K instances near it. The following figure shows an example of K-NN.

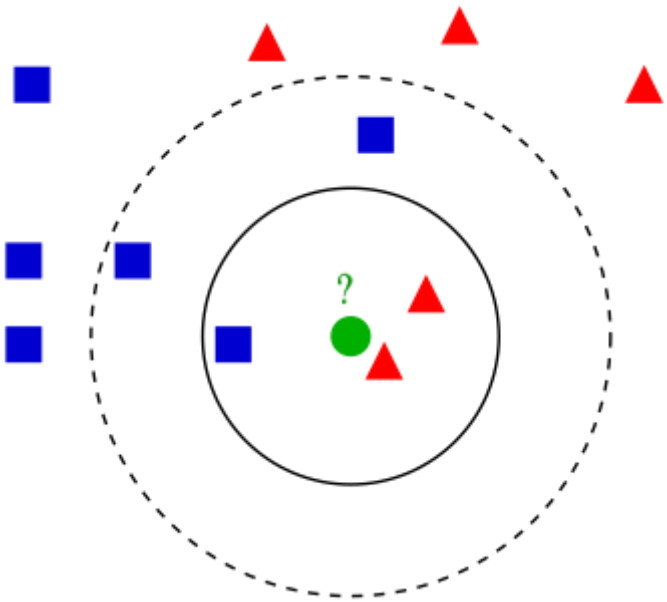

Figure 2: K-NN example (Wikipedia, 2022)

As can be seen in the example, if K=3 the new green instance will be red as there are two red instances and one blue instance as the three closest instances. But if k=5, the result would be that the new instance would be blue.

3 Practical Implementation

This practice consists of working with the sklearn digits dataset where a series of digits of 8x8 pixels are given, and where an analysis, a pre-processing using PCA and a post-classification using the K-NN algorithm have to be done.

3.1 Data analysis

In order to work with the digits dataset, the first thing to do is to load the dataset. Once we have done the load, we explore the dataset making a series of prints such as the number of simplexes, the shape, the target variable (which is the number resulting from the digits) and other data.

After loading the dataset, the first thing that we did was to show *DESCR*, which is an attribute that containing metadata about the dataset. It shows the number of digits, and the number of pixels per digit which is 64 (8x8 Pixels)
Also, it says that the digits are in greyscale where 0 is white and 16 is dark, and all the numbers in between are different tones of grey.

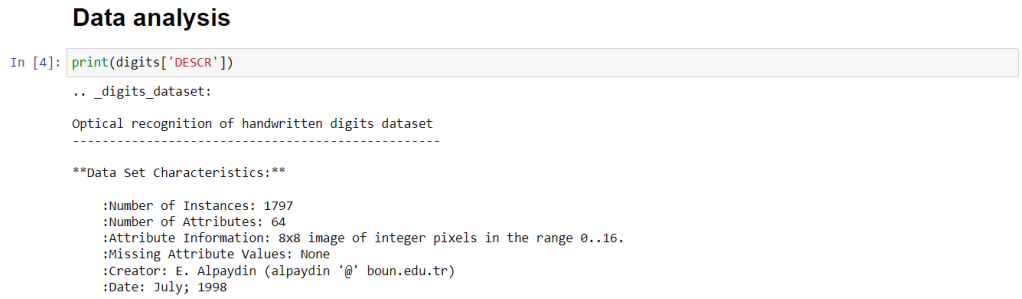

Figure 3: DESCR attribute in the digits dataset.

After that, we showed an example of a digit to show how data is presented. In this case, we showed digit 1021, but this analysis could have been done with any other digit in the dataset.

Digit Visualization Example

```
In [12]: print("Target value: ", digits.target[1021])
         print("Digit array: ", digits.images[1021])
         plt.imshow(digits.images[1021], cmap=plt.cm.gray_r, interpolation='nearest')
         plt.show()

Target value:  5
Digit array:  [[ 0.  2. 16. 16. 16. 16.  4.  0.]
 [ 0.  4. 16.  6.  8.  7.  1.  0.]
 [ 0.  4. 16.  7.  2.  0.  0.  0.]
 [ 0.  4. 16. 16. 16.  6.  0.  0.]
 [ 0.  0.  5.  4. 10. 15.  0.  0.]
 [ 0.  0.  0.  0.  1. 14.  6.  0.]
 [ 0.  2. 14.  4.  4. 16.  8.  0.]
 [ 0.  3. 13. 16. 16. 15.  1.  0.]]
```

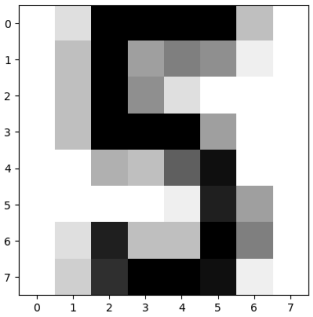

Figure 4: Digit example from the digits dataset

As can be seen from the image, the target value of digit[1021] is a 5. The digits.images[1021] shows the matrix of numbers representing the digit (Remember that a 0 is white as bigger is the number darker it is up to 16).
Then, we plot the digits.images[1021] and we can see that it is a 5 as the target value said.
We also showed some extra metadata such as the number of samples or the target values for all digits.

Digit Metadata

```
In [13]: print(digits.images.shape)
         print(digits.data.shape)
         print(digits.target)

(1797, 8, 8)
(1797, 64)
[0 1 2 ... 8 9 8]
```

Figure 5: Showing digits dataset metadata

As can be seen from the previous image, there are 1797 digits of 64 values each, as described by the DESCR attribute.

3.2 Preprocessing

After doing an initial data analysis, we had to start doing a previous preprocessing prior to the KNN classification. As stated in the statement, we first had to do a data split with 75% of train data and a 25% of test. The purpose of dividing the data in test and train is to be able to use accurate data models. In our case, we divided it in the percentages stated in the description of the project and we added some randomness to make it more accurate.

Data Analysis

Data Split

In [22]:
```
##- split the data 75% train and 25% test subsets
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(X, Y, test_size = 0.25, random_state=42, stratify=Y)
```

In [23]:
```
print("(Num images, Image size)")
print(x_train.shape)
print(x_test.shape)
print("(Predicted number)")
print(y_train.shape)
print(y_test.shape)
```

```
(Num images, Image size)
(1347, 64)
(450, 64)
(Predicted number)
(1347,)
(450,)
```

Figure 6: Data split in train and test

As can be seen from the image, we split the data and then we show how the train and test X and Y add up to 1797. The train data has 1797 * 0,75 values which is 1347 digits, and for the test it is 1797*0,25 = 450.
Then, what we do is to normalize the data to obtain a zero mean and a unit variance, that is, variance can only have a value of 1 and 0.

Data Normalizationby zero mean and unit variance

In [24]:
```
##- Data normalization by zero mean and unit variance
from sklearn import preprocessing

scaler = preprocessing.StandardScaler().fit(x_train)
x_train_scaled = scaler.transform(x_train)
```

In [25]:
```
print("Showing that scaled data has zero mean and unit variance:")
print("Mean")
print(x_train_scaled.mean(axis=0))
print("Variance")
print(x_train_scaled.std(axis=0))
```

```
Showing that scaled data has zero mean and unit variance:
Mean
[ 0.00000000e+00 -8.43176061e-17  8.16801052e-17 -2.03252411e-16
  1.10816248e-16  2.53859459e-17 -1.68882478e-16  1.51058742e-16
  1.05293980e-17  2.08527413e-16  1.89735219e-16  7.28609617e-17
 -2.12153977e-16 -4.12521621e-17  3.24742295e-16  3.82643682e-16
  1.32019282e-16  2.95400098e-16 -2.90125096e-17 -1.36820358e-16
 -3.68129700e-17  2.30781327e-17 -7.47566654e-17  2.26701442e-16
 -3.00968728e-16  2.91155370e-17  6.05800982e-17 -4.45078273e-17
  6.92343980e-17 -6.98937732e-17 -2.15615697e-16  1.55664065e-16
  0.00000000e+00 -2.96224317e-16 -6.69265847e-17  1.70943025e-16
 -1.63195367e-16 -1.91878189e-16 -3.51117304e-17  0.00000000e+00
  3.10565728e-16  1.70448494e-16 -5.52226746e-17 -8.72023727e-17
  3.84086065e-17  5.82722850e-17 -1.25610979e-16 -1.43764403e-16
 -1.55694974e-16 -3.36281362e-17  4.82168129e-17  1.47535205e-17
 -1.52315676e-16  5.15136890e-18  3.29028234e-16  3.29069445e-16
 -3.40309732e-16  1.61876616e-16  3.99746226e-18 -3.68096216e-16
 -8.77793260e-17 -6.21461144e-17 -1.47205518e-16 -2.50562583e-17]
Variance
[0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 0. 1. 1. 1. 1. 1. 1. 1. 0. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

Figure 7: Data normalization with zero mean and unit variance

To do the pre-processing, we use the `StandardScaler()` and we first do a `fit()` and then a `transform().` After doing this, we print the values of the mean and from the array on the previous figure, we see that all the numbers are very small and almost close to 0. Then, we print the variance, and it can be seen that all the values are either 1 or 0. So the pre-processing was correct. After doing this, we had to apply the scaling on the test data. To do that, we used the same scaler of the train data to the test data. The following figure shows it:

```
In [125]: # apply scaling on testing data having into account x_train statistics
          x_test_scaled = scaler.transform (x_test)
```

Figure 8: Applying scaling on the test data

Once the normalization and the scaling are done, we proceeded with the PCA analysis. For this, we used the `PCA()` function provided by sklearn. Concretely, we passed as parameter a 0.95 indicating that we want components up to a 95% of accuracy. Then, we applied the PCA by using the `fit()` function. After that, we showed some data to screen.

PCA Analysis

```
In [171]: ##- Dimensionality reduction: PCA and SVD analysis
          from sklearn.decomposition import PCA
          from sklearn.decomposition import TruncatedSVD

          ##- PCA Analysis
          pca = PCA(n_components=0.95)
          pca.fit(x_train_scaled)
          size = numpy.arange(10, pca.explained_variance_ratio_.shape[0])

          print("Variance: ", pca.explained_variance_ratio_[0:5])
          print("Variance ratio: ",pca.explained_variance_[0:5])
          print("Singular Values: ", pca.singular_values_[0:5])

          Variance:  [0.11918083 0.09668681 0.083441   0.06702611 0.04851085]
          Variance ratio:  [7.27543205 5.90227709 5.09368231 4.09163026 2.96136043]
          Singular Values:  [98.95823125 89.13172817 82.80154825 74.21141643 63.13470628]
```

Figure 9: PCA computation

As can be seen from the result, we print the first 5 components, their variances, variance percentage and its singular values. As can be seen, the variance ratio is decreasing. This is because in PCA the first components have more importance, thus, they have a higher variance. As components get less important in means of variance, they are placed latter in the array.
Then, what we did was to compute and plot the accumulated variance to know the number of components needed to reach a 90% of the variance.

```
In [210]: num_components_needed = 0
found = 0
curr_total_variance = 0
cumulative_variance = numpy.empty(pca.explained_variance_ratio_.shape[0])
last_accumulated_variance = 0
# Computing cumulative variance
for i in range(0,pca.explained_variance_ratio_.shape[0]):
    cumulative_variance[i] = curr_total_variance + pca.explained_variance_ratio_
    curr_total_variance += pca.explained_variance_ratio_[i]
    if curr_total_variance > 0.9 and found == 0:
        num_components_needed = i + 1
        found = 1
        last_accumulated_variance = curr_total_variance

print("Num components needed for a", last_accumulated_variance, "% accuracy: ",
##- the PCA explains a part of the variance. Plot the cumulative variance to get
## components we need.
plt.title('PCA')
plt.xlabel('Number of Components')
plt.ylabel('Acucumuled Variance')
plt.plot( cumulative_variance, label = 'Cumulative variance')
plt.legend()
plt.show()
```

Figure 10: Computation of needed components in PCA

Basically, the previous code shows how we compute the accumulated variance and how we plot it using matplotlib.
The output of the code of the previous figure is the following:

```
Num components needed for a 0.9025496322838922 % accuracy:  31
```

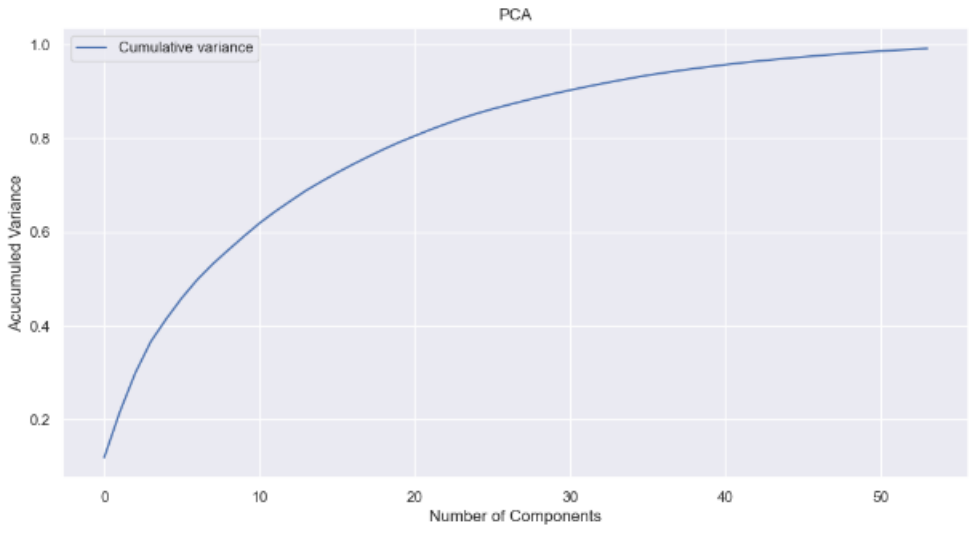

Figure 11: Plot with components needed in PCA

As can be seen from the previous figure, when it reaches 31 components, it has more than the 90% of the variance. Therefore, for a study, 31 components could be enough.

Then, we had to do the SVD analysis. Similarly, we used the `sklearn` library of python. To compute the SVD we passed the number of components found

in PCA and the scaled train data. Then we did the `fit()` to compute the SVD. After computing the SVD, we can see that the singular values are the same as those in PCA and that the variance ratio is the same too. To compare the result with PCA, we added all the variance ratio for 31 components, and we can see that the result is the same as in PCA.
Therefore, both with SVD and with PCA, the components and the variance ratios are the same.

SVD Analysis

In [211]:
```
##- SVD Analysis
tsvd = TruncatedSVD(n_components=(num_components_needed)).fit(x_train_scaled)
## - Same analysis as PCA: you have to obtain the best number of components for SVD.
print("Variance: ", tsvd.explained_variance_ratio_[0:5])
print("Singular Values: ", tsvd.singular_values_[0:5])
print("Accumuled variance in % for", num_components_needed," components is: ", tsvd.explained_variance_ratio_.sum())
```

```
Variance:  [0.11918083 0.09668681 0.083441   0.06702611 0.04851085]
Singular Values:  [98.95823125 89.13172817 82.80154825 74.21141643 63.13470628]
Accumuled variance in % for 31  components is:  0.9025437196313869
```

Figure 12: Computing the SVD

3.3 K-NN Classification

Once the SVD and the PCA were computed, we could apply the KNN. For the KNN, we did two analyses. The first analysis we did was on the effect of changing the number of neighbours (K), on the non-pre-processed data and we plotted the accuracy.
And the second analysis was on the effect of changing K on the pre-processed data.

Not Pre-processed Data

The code of the first experiment is:

Different k analysis Not Preprocessed KNN

In [116]:
```
##- k neighbours analysis: different clsssifier performances varyng the number of k

# Setup arrays to store train and test accuracies
neighbors = numpy.arange(1, 20)
train_accuracy = numpy.empty(len(neighbors))
test_accuracy = numpy.empty(len(neighbors))
cross_fold_accuracy = numpy.empty(len(neighbors))
deviation = numpy.empty(len(neighbors))

# Loop over different values of k
for i, k in enumerate(neighbors):
    # Setup a k-NN Classifier with k neighbors: knn
    knn = KNeighborsClassifier(n_neighbors=k)
    # Fit the classifier to the training data
    knn.fit(x_train, y_train)
    #Compute accuracy on the training set
    train_accuracy[i] = knn.score(x_train, y_train)
    #Compute accuracy on the testing set
    test_accuracy[i] = knn.score(x_test, y_test)
    cv_scores = cross_val_score(knn, x_train, y_train, cv=10)
    cross_fold_accuracy[i] = numpy.mean(cv_scores)
    deviation[i] = cv_scores.std()
```

In [117]:
```
# Generate plot
plt.title('k-NN: Varying Number of Neighbors')
plt.plot(neighbors, test_accuracy, label = 'Testing Accuracy')
plt.plot(neighbors, train_accuracy, label = 'Training Accuracy')
plt.plot(neighbors, cross_fold_accuracy, label = 'Cross Validation score')

plt.legend()
plt.xlabel('Number of Neighbors')
plt.ylabel('Accuracy')
plt.show()
```

Figure 13: Code generating the K-NN for different amount of neighbours

As can be seen from the code, we first do a for lop to compute the train and test accuracy.

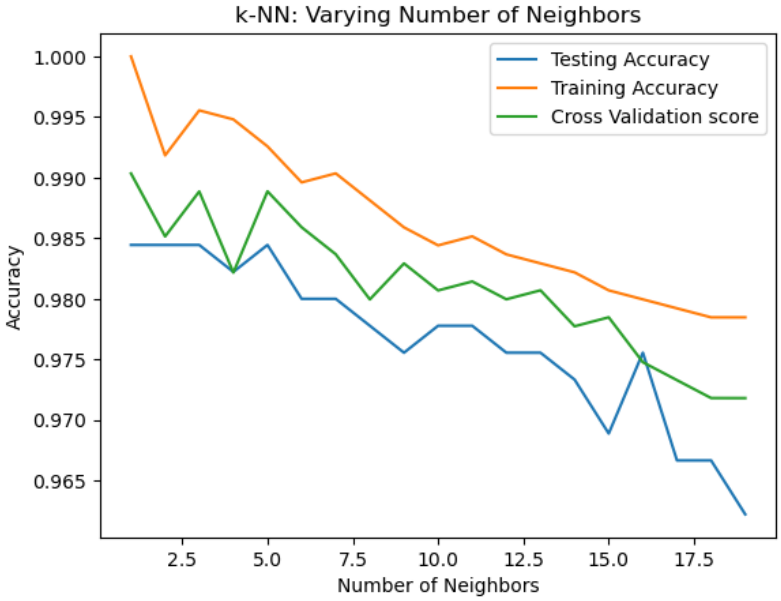

Figure 14: Plot showing the accuracy in train, test and cross validation score with different k

```
In [28]: plt.title('k-NN: Varying Number of Neighbors')
         plt.xlabel('Number of Neighbors')
         plt.ylabel('Deviation')
         plt.plot(neighbors, deviation, label = 'Deviation')
         plt.legend()
         plt.show()
```

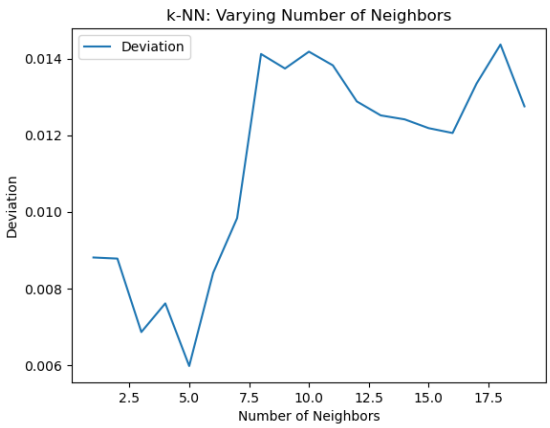

Figure 15: Plot showing the deviation evolution for the increase of k

As can be seen from the two previous plots, the training accuracy is always a bit better than the testing accuracy. However, the important thing to notice is that all three accuracies are decreased as the number of neighbours increase. While at the same time, the deviation seems to increase as neighbours increase.

Pre-processed Data

The second experiment consisted on the same thing as the previous one, but, using the pre-processed data. Therefore, the code for the plots is the same but only changing the trained X and Y.
The results are shown in the following figures:

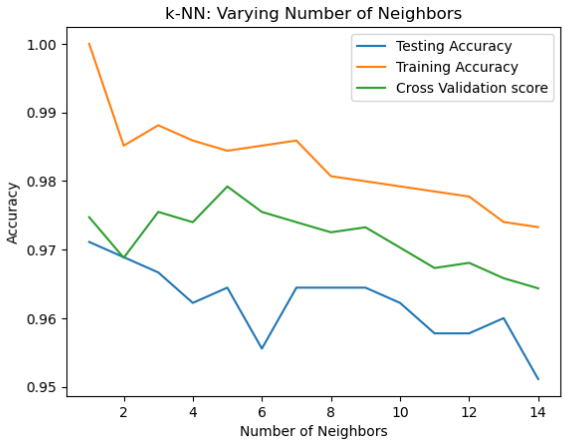

Figure 16: Plot showing the K-NN accuracy of train, test and cross score for processed data

```
In [31]: plt.title('k-NN: Varying Number of Neighbors')
         plt.xlabel('Number of Neighbors')
         plt.ylabel('Deviation')
         plt.plot(neighbors, deviation, label = 'Deviation')
         plt.legend()
         plt.show()
```

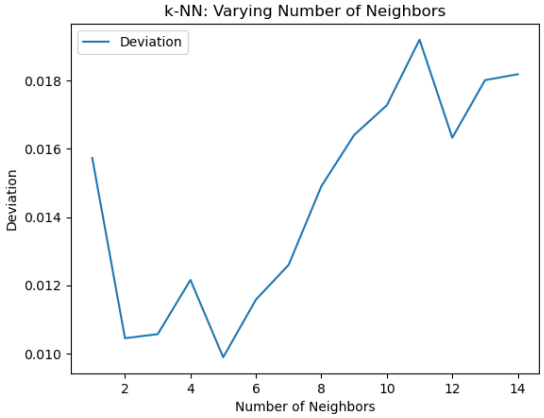

Figure 17: Plot showing the evolution of deviation for the increase of k in the preprocessed data

4 Conclusion

This project has been useful for several reasons. Mainly because we have been able to put into practice several theoretical concepts seen in class such as the instance-based learning with K--NN classifiers. Or some pre-processing algorithms of dimension reduction.

Apart from this, we have also been able to draw some conclusions about the algorithms we have dealt with such as K-NN. A first conclusion is that as the number of neighbours increases, the precision of the algorhythm seems to decrease. One of the possible reasons for this is that too distant neighbours are taken into account. However, on the other hand, the higher the number of neighbours, the smoother the boundary prediction. Not only that, but also, the deviation increases as the number of neighbours increases. Or at least that is what happens in our experiment.

Another conclusion we can draw is that doing the pre-processing and dimension reduction by doing PCA, the results are quite similar while at the same time, we are computing data with less dimensions. This indicates that despite reducing dimensions with PCA, if we take into account the most important principal components, we do not lose much important data while reducing the computational power needed to do the computations.

As a more personal conclusion, the realization of this practice has helped me to understand more the concepts of class with practical tools such as the scikit Python library.

5 Bibliography

Wikipedia. (2022). *k-nearest neighbors algorithm*. Retrieved from https://upload.wikimedia.org/wikipedia/commons/thumb/e/e7/KnnClassification.svg/330px-KnnClassification.svg.png

Wikipedia. (2022). *PCA of a multivariate Gaussian distribution*. Retrieved from https://en.wikipedia.org/wiki/Principal_component_analysis