Assignment 3

Marion Rosec

Decembre 2023

Exercise 1

a)

We had to adapt the code of the last assignment as we needed to transpose the data for the PCA implementation to work. The visualization of 2D data after PCA was "inverted" and we needed to add a minus to the covariance matrix in the PCA function to solve that problem. With this, we obtained the following 2D representation of the data :

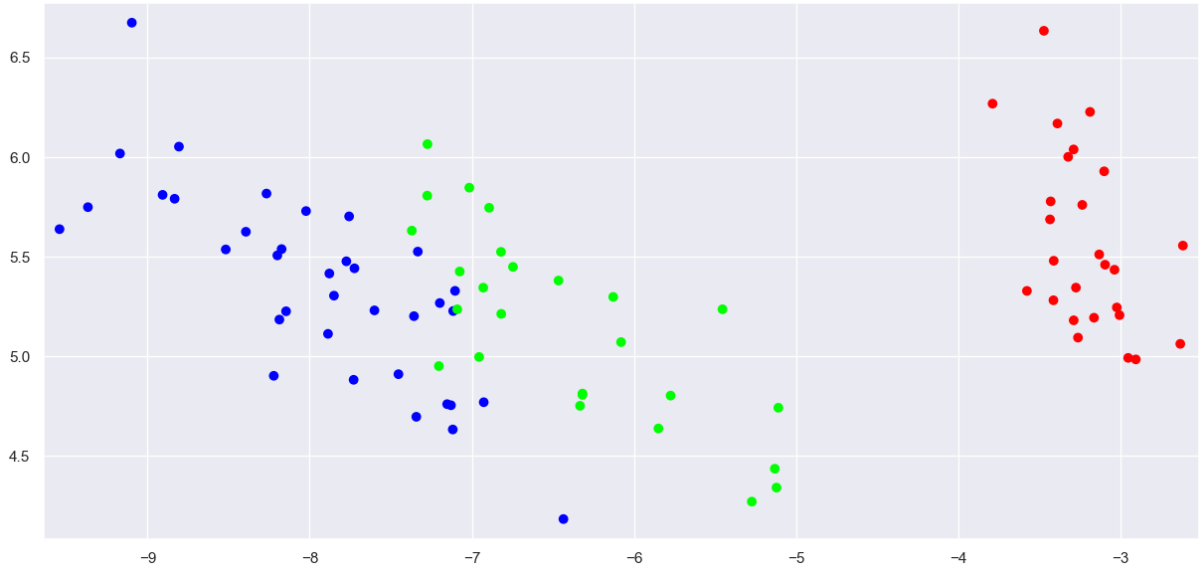

Figure 1 – 2D Visualization of training set by labels

Python

```python
def __PCA(data):
    data = np.transpose(data)
    data_cent = data - np.mean(data)
    data_cov = -np.cov(data_cent)
    PCevals, PCevecs = np.linalg.eig(data_cov)
    return PCevals, PCevecs
```

```python
def __transformData(features, PCevecs):
    return np.dot(features, PCevecs[:, 0:2])


PCevals, PCevecs = __PCA(trainingFeatures)
trainingFeatures2D = __transformData(trainingFeatures, PCevecs)
validationFeatures2D = __transformData(validationFeatures, PCevecs)
testingFeatures2D = __transformData(testingFeatures, PCevecs)
print("shape training = ", trainingFeatures2D.shape)
print("shape validation = ", validationFeatures2D.shape)
print("shape testing = ", testingFeatures2D.shape)
```

Python

```
shape training =  (88, 2)
shape validation =  (29, 2)
shape testing =  (31, 2)
```

b) and c)

For the kNN, we first need to compute the distance between each point of the validation set and all the points of the training set. We then obtain an array of distance to the points of the training set for each points in the validation set. For each of the points, we then need to find the n-best distances in the array and get the indexes corresponding to the points in the training that gives the best distances. Thanks to the indexes, we can find the labels corresponding to these points and keep the labels that appears the most as the predicted label for our point in the validation set. After doing this process for all the point (and keeping these predicted label in a list) we can compute the accuracy by comparing the real labels for the validation data set, and the ones predicted by the kNN implementation. We can now try several different numbers of neighbors to see which value of n gives the best accuracy :

Python

```
accuracy =  0.9310344827586207
accuracy =  0.9655172413793104
accuracy =  1.0
accuracy =  1.0
accuracy =  1.0
```

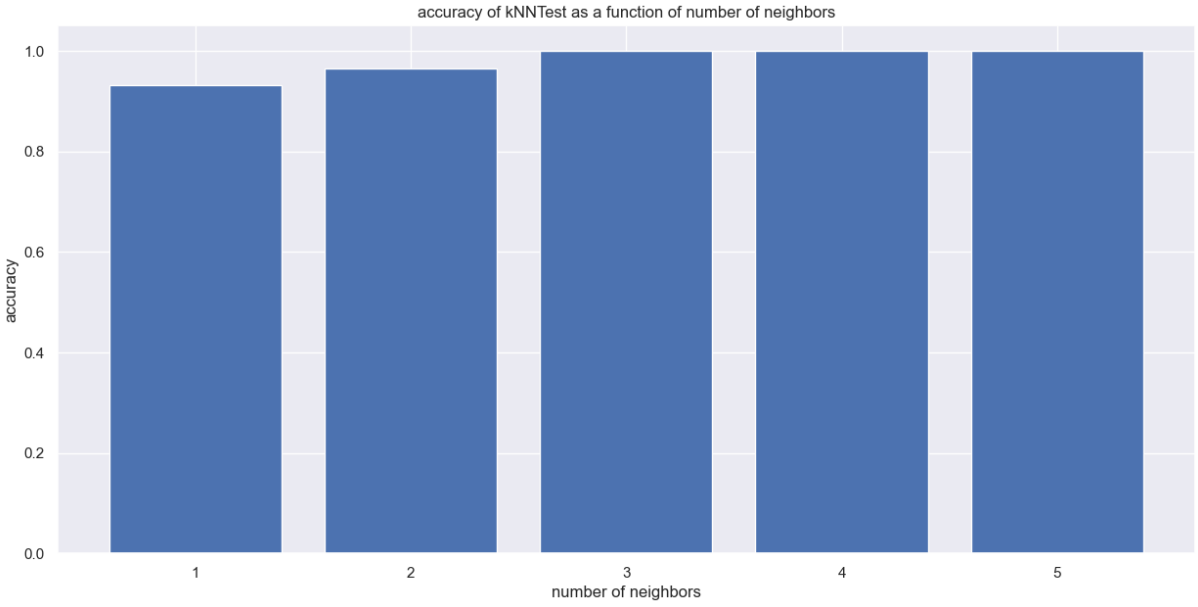

Figure 2 – Accuracy as a function of the number of neighbors used in the kNN test

We can see that we have the best result (the best accuracy) with a number of neighbors superior or equal to 3.

Python

```python
def __kNNTest(
    trainingFeatures2D,
    trainingLabels,
    n_neighbors,
    validationFeatures2D,
    validationLabels,
):
    predicted_labels = []
    for i in range(len(validationFeatures2D)):
        distance = np.linalg.norm(validationFeatures2D[i] - trainingFeatures2D, axis=1)
        idx = np.argsort(distance)[:n_neighbors]
        predict_label = np.bincount(trainingLabels[idx]).argmax()
        predicted_labels.append(predict_label)
    predicted_labels = np.array(predicted_labels)
    accuracy = np.mean(validationLabels == predicted_labels)
    return accuracy
```

d)

For the Random Forest classification, we can use the existing implementation from sklearn and apply it on our data to be able to use it later for the visualization. We can also change the parameters of the RandomForestClassifier to fit our iris dataset the best. In this case, we set the number of tree to 150, and the maximum depth to 20 because it is not necessary until all leaves are pure.

Python

```python
def __randomForests(trainingFeatures2D, trainingLabels):
    predictor = RandomForestClassifier(
        n_estimators=150,
        max_depth=20,
```

```
    )
    predictor.fit(trainingFeatures2D, trainingLabels)
    return predictor

predictor = __randomForests(trainingFeatures2D, trainingLabels)
```

e)

To visualize the data, we can firstly use the existing kNN implementation from sklearn, then we create a function that will create a plot to visualize our data no matter what the classification may be. Here, we adapted the code from the link given in the notebook to our function. We can then apply to our data the 2 classification (kNN and Random forest) and visualize them. For the kNN, we set the number of neighbors to 3 as we found that it was optimal in question a). For both of the plots, the colors of the points represent the real labels of the test set and the colored areas represent the decisions boundaries which define the predicted labels for the test data.

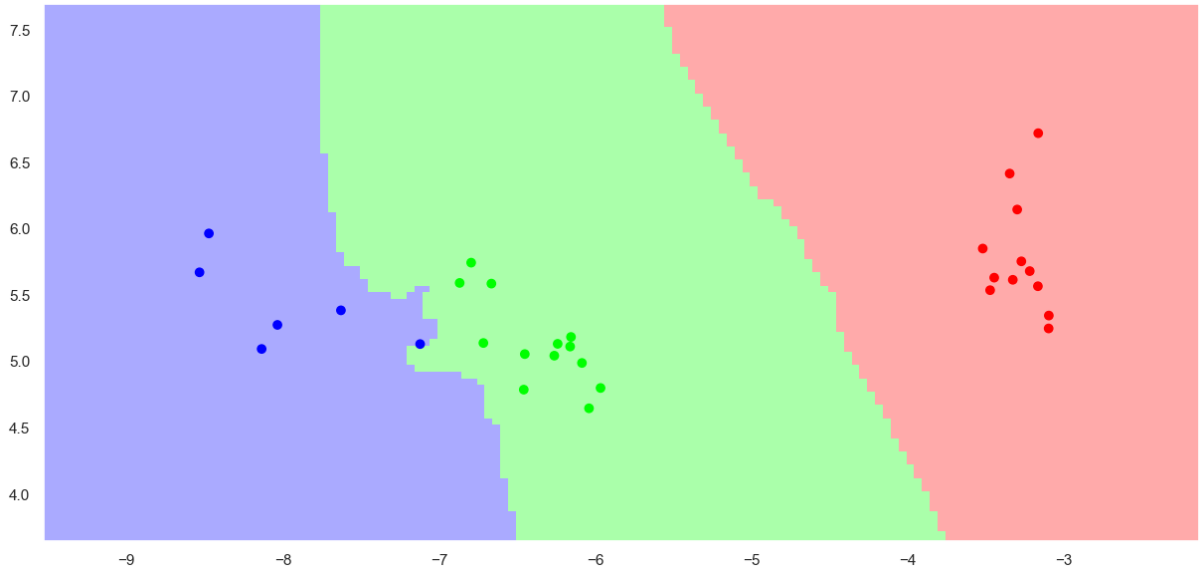

Figure 3 – Visualization of the data with kNN classification

For the kNN, we can see that our data, which separated into 3 differents clusters, are fairly well disperse in the decisions areas. Therefore, we can think that the classification fits well to the data; the kNN algorithm gives good label predictions with 3 neighbors.

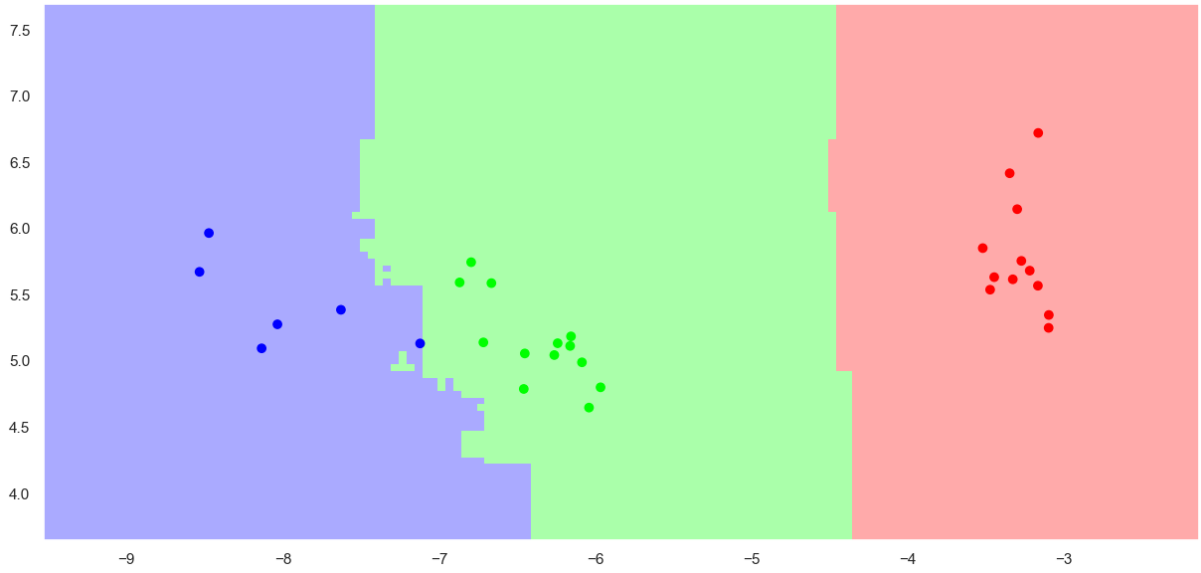

Figure 4 – Visualization of the data with Random Forest classification

For the random forest classification, the results are a bit different from those of the kNN, but we can still see that the decision areas fits the real label of the data. We can then say that the random forest classification gives good prediction of label for the testing set. However, this plot is not unique as the trees used for the final data are chosen randomly and if we reapply the Random forest classifier to our data, the visualization may differ a bit.

Both plots are pretty similar, so we cannot really say which one is better just from looking at them.

Python

```
def __kNN(trainingFeatures2D, trainingLabels, n_neighbors):
    predictor = KNeighborsClassifier(n_neighbors)
    predictor.fit(trainingFeatures2D, trainingLabels)
    return predictor


def __visualizePredictions(predictor, features, referenceLabels):
    plt.figure()
    cmap_light = ListedColormap(["#FFAAAA", "#AAFFAA", "#AAAAFF"])
    cmap_bold = ListedColormap(["#FF0000", "#00FF00", "#0000FF"])
    h = 0.05
    y = referenceLabels

    # Plot the decision boundary. For that, we will assign a color to each
    # point in the mesh [x_min, x_max]x[y_min, y_max].
    x_min, x_max = features[:, 0].min() - 1, features[:, 0].max() + 1
    y_min, y_max = features[:, 1].min() - 1, features[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
    Z = predictor.predict(np.c_[xx.ravel(), yy.ravel()])

    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    plt.figure()
    plt.pcolormesh(xx, yy, Z, cmap=cmap_light)

    # Plot also the training points
    plt.scatter(features[:, 0], features[:, 1], c=y, cmap=cmap_bold)
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())
    plt.show()
```

```
k = 3
kNNPredictor = __kNN(trainingFeatures2D, trainingLabels, k)
RFPredictor = __randomForests(trainingFeatures2D, trainingLabels)
__visualizePredictions(kNNPredictor, testingFeatures2D, testingLabels)
__visualizePredictions(RFPredictor, testingFeatures2D, testingLabels)
```