Labwork 3 – Report

Member:
Phan Manh Tung – USTHBI8-160
Vu Tuan Phong – USTHBI8-139

T2 – Classification I

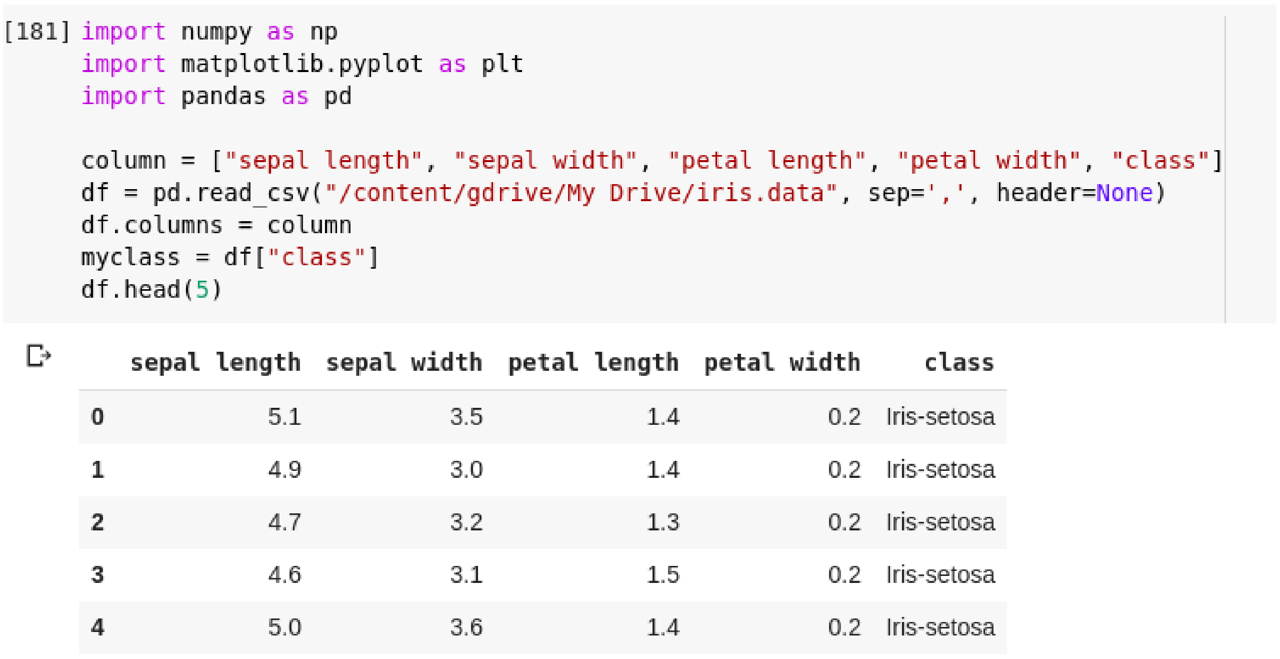
Data: For this labwork, we choose 2 dataset from UCI Machine Learning Repository namely Iris dataset and Breast Cancer Wisconsin dataset.

Tool: Python is our chosen programming language and we opt for google colab (a free python interactive environment offered by Google) for strong computing power and convenient coding.

I) Perceptron

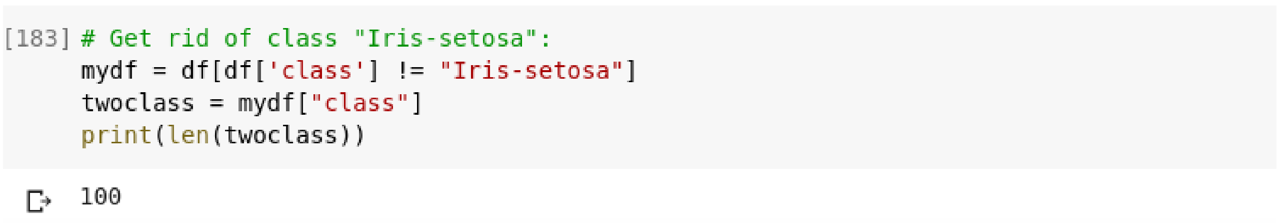
<u>1.1 Iris dataset:</u>

In this problem, we attempt to implement the model from scratch.
First, we prepare the demanding input for the model. Iris dataset has 3 classes, which is not suitable for our binary model. Therefore, we eliminate class Iris-setosa, leaving the dataset with 100 sample divided into 2 classes namely, Iris-virginica, Iris-versicolor.



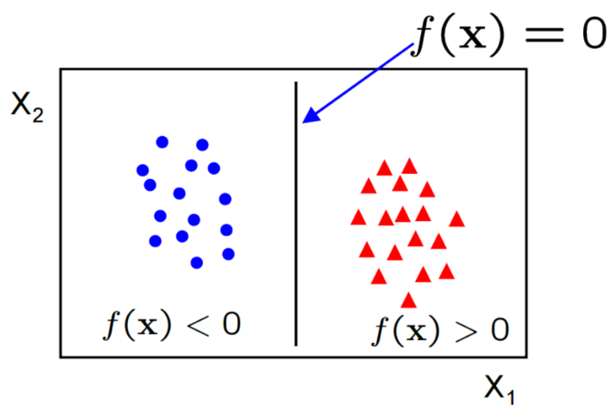
```
[181] import numpy as np
      import matplotlib.pyplot as plt
      import pandas as pd

      column = ["sepal length", "sepal width", "petal length", "petal width", "class"]
      df = pd.read_csv("/content/gdrive/My Drive/iris.data", sep=',', header=None)
      df.columns = column
      myclass = df["class"]
      df.head(5)
```

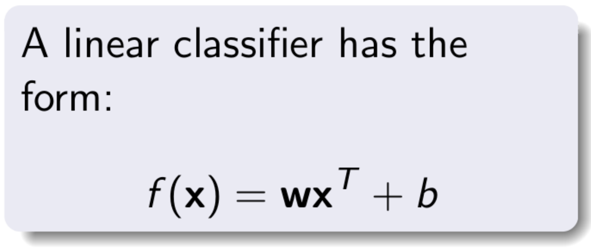
	sepal length	sepal width	petal length	petal width	class
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa



```
[183] # Get rid of class "Iris-setosa":
      mydf = df[df['class'] != "Iris-setosa"]
      twoclass = mydf["class"]
      print(len(twoclass))
```

```
100
```

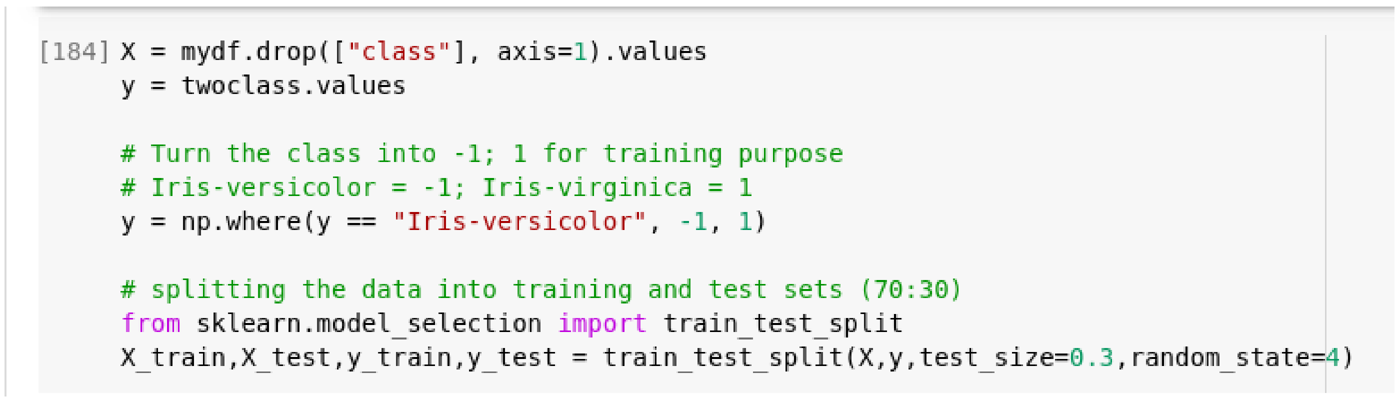




A linear classifier has the form:

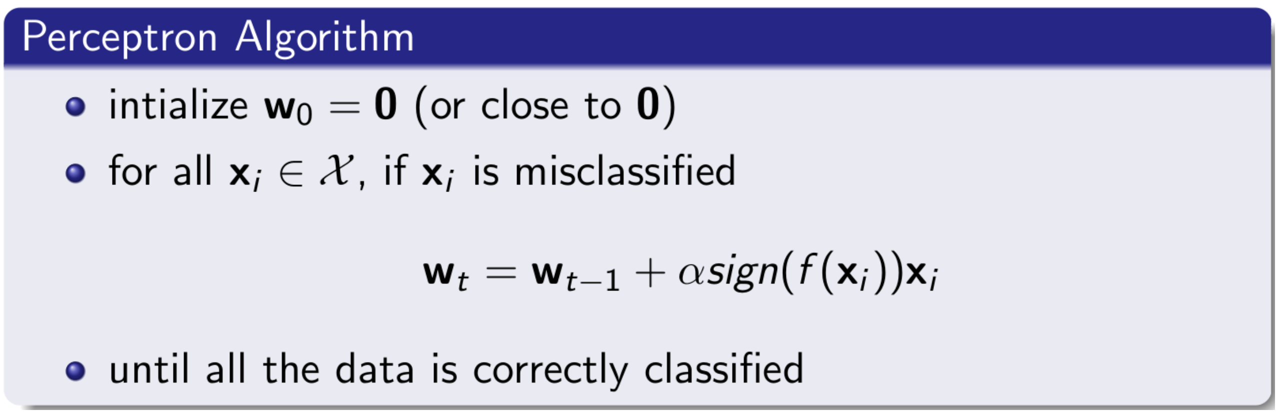
$$f(\mathbf{x}) = \mathbf{w}\mathbf{x}^T + b$$

Because our activation function of Perceptron algorithm is the sign of f(x) – the hyperplane, so that we also have to change the labels of the dataset into values 1 and -1 (-1 is Iris-versicolor and 1 is Iris-virginica).
Then we divide the dataset into train/test set with the ratio of 70:30.



```
[184] X = mydf.drop(["class"], axis=1).values
      y = twoclass.values

      # Turn the class into -1; 1 for training purpose
      # Iris-versicolor = -1; Iris-virginica = 1
      y = np.where(y == "Iris-versicolor", -1, 1)

      # splitting the data into training and test sets (70:30)
      from sklearn.model_selection import train_test_split
      X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=0.3,random_state=4)
```

Below part is our main implementation of the Perceptron algorithm. We follow the rule given on the slide.

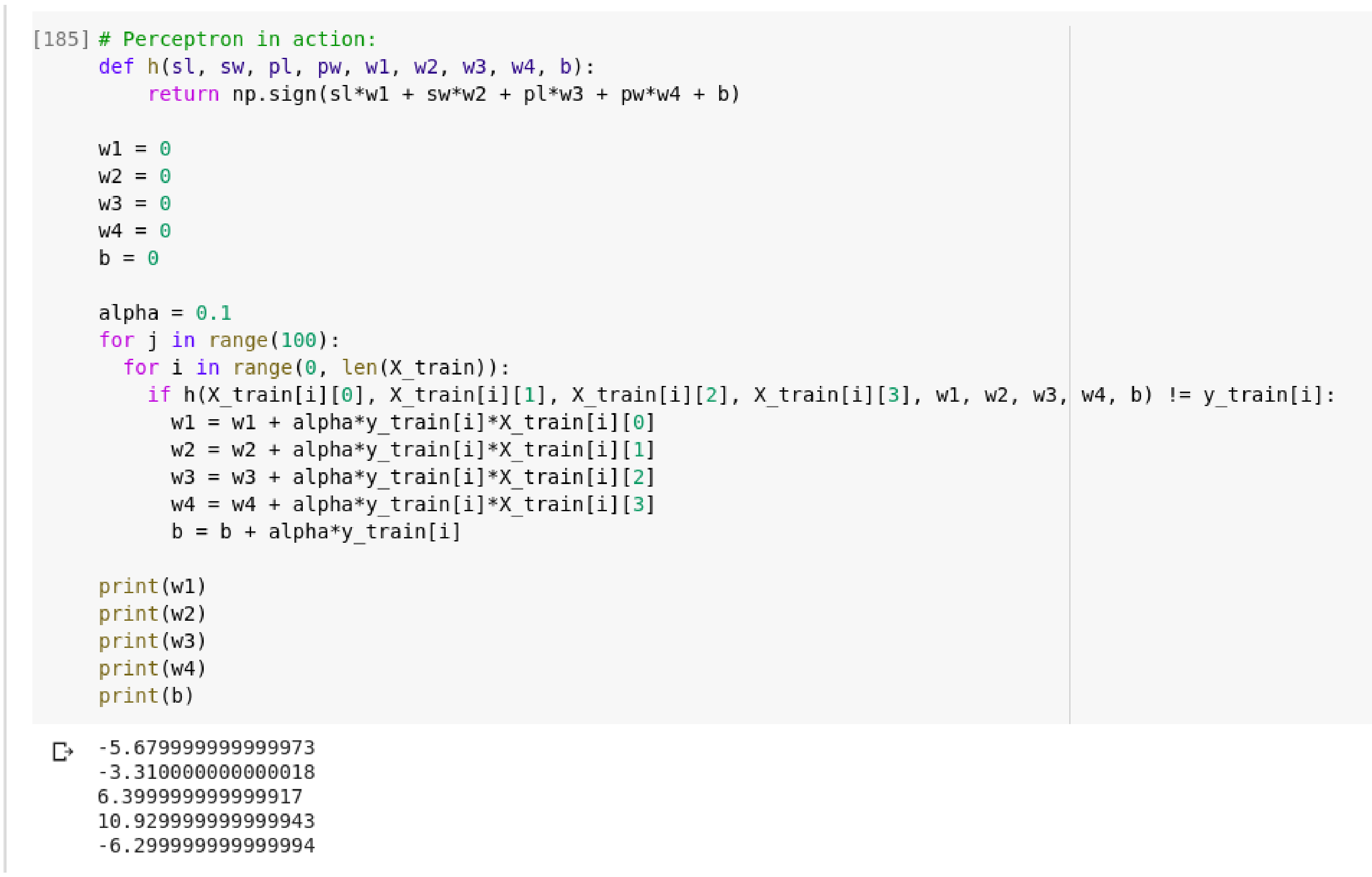


Perceptron Algorithm

- intialize $\mathbf{w}_0 = \mathbf{0}$ (or close to $\mathbf{0}$)
- for all $\mathbf{x}_i \in \mathcal{X}$, if $\mathbf{x}_i$ is misclassified

$$\mathbf{w}_t = \mathbf{w}_{t-1} + \alpha sign(f(\mathbf{x}_i))\mathbf{x}_i$$

- until all the data is correctly classified



```
[185] # Perceptron in action:
     def h(sl, sw, pl, pw, w1, w2, w3, w4, b):
         return np.sign(sl*w1 + sw*w2 + pl*w3 + pw*w4 + b)

     w1 = 0
     w2 = 0
     w3 = 0
     w4 = 0
     b = 0

     alpha = 0.1
     for j in range(100):
       for i in range(0, len(X_train)):
         if h(X_train[i][0], X_train[i][1], X_train[i][2], X_train[i][3], w1, w2, w3, w4, b) != y_train[i]:
           w1 = w1 + alpha*y_train[i]*X_train[i][0]
           w2 = w2 + alpha*y_train[i]*X_train[i][1]
           w3 = w3 + alpha*y_train[i]*X_train[i][2]
           w4 = w4 + alpha*y_train[i]*X_train[i][3]
           b = b + alpha*y_train[i]

     print(w1)
     print(w2)
     print(w3)
     print(w4)
     print(b)
```

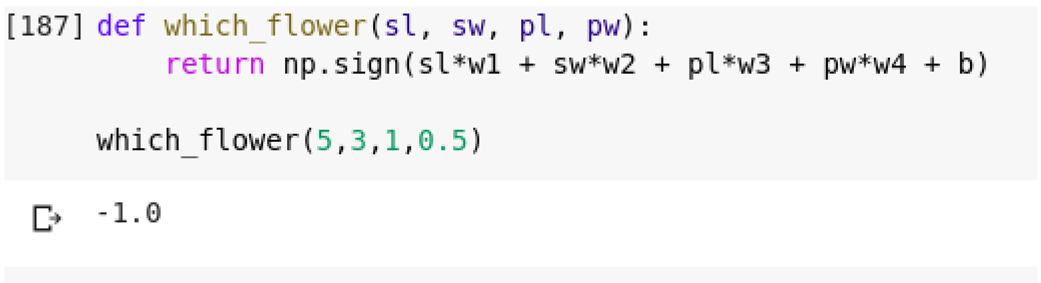
```
-5.679999999999973
-3.310000000000018
6.399999999999917
10.929999999999943
-6.299999999999994
```

The h() function is our activation function, which returns the sign of the hyperplane (value 1 or -1).

Because Iris dataset has 4 attributes: sepal length, sepal width, petal length, petal width. The hyperplane f(x) will contain 4 weights w1, w2, w3, w4 and bias value b. We initialize all 5 values at 0.

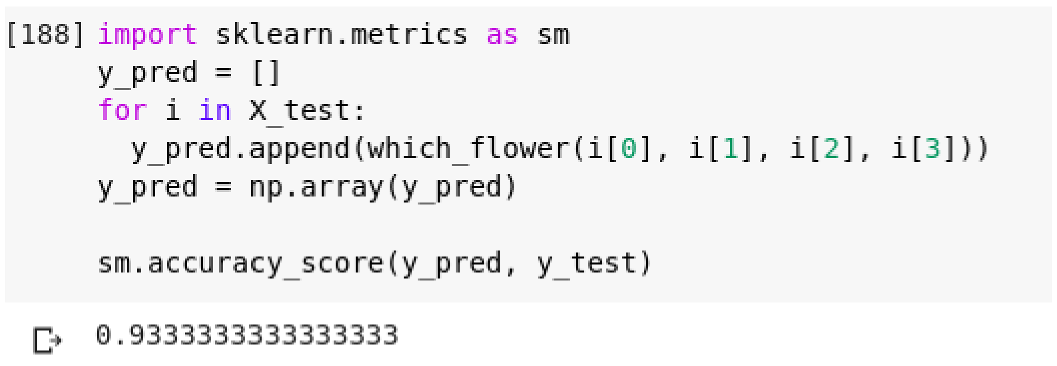
The learning rate (alpha) is set at 0.1

In the main for-loop, we have 100 iterations which is not a large value. Looping through the training set, for each misclassified point, we use that point to update our weights.
Then we use our model to make a very first prediction:



```
[187] def which_flower(sl, sw, pl, pw):
          return np.sign(sl*w1 + sw*w2 + pl*w3 + pw*w4 + b)

      which_flower(5,3,1,0.5)
```

```
-1.0
```



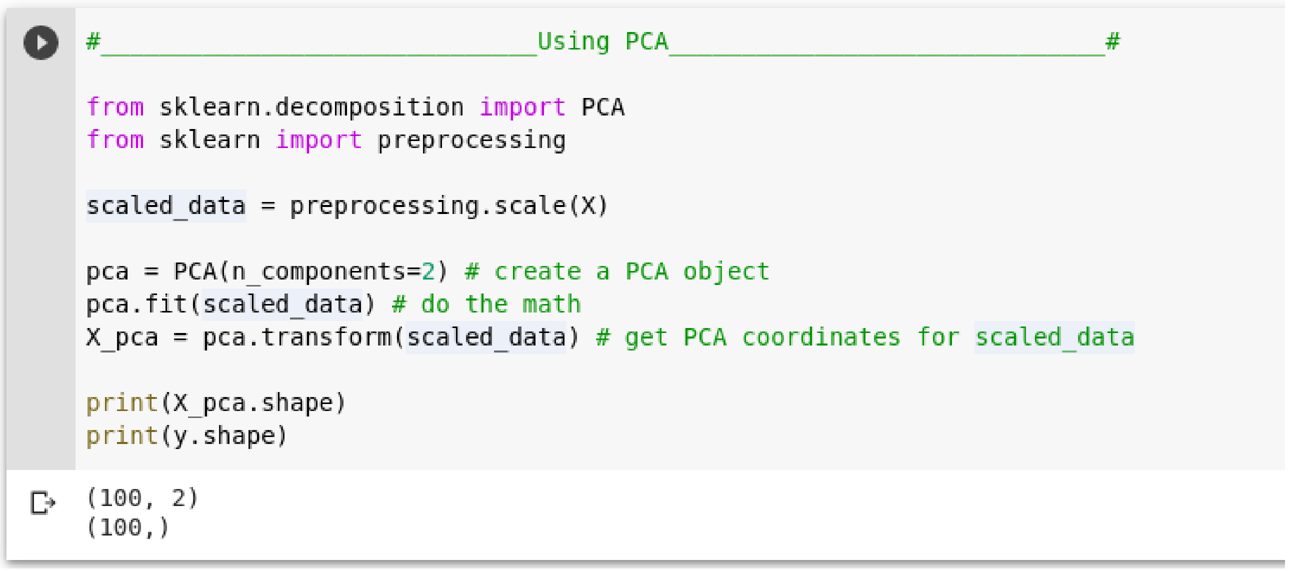
```
[188] import sklearn.metrics as sm
      y_pred = []
      for i in X_test:
        y_pred.append(which_flower(i[0], i[1], i[2], i[3]))
      y_pred = np.array(y_pred)

      sm.accuracy_score(y_pred, y_test)
```

```
0.9333333333333333
```

Afterwards, we use obtained weights to make predictions on the test set, then use the metric accuracy_score to compare with the y_test labels. Our model is 93.33% accurate on the testing set, which is a great result.

1.2 Using PCA for 2D visualization:

For easier visualization, we apply the PCA technique to reduce our 4-dimensional Iris dataset into a 2-dimensional dataset.



```
#______________________________Using PCA______________________________#

from sklearn.decomposition import PCA
from sklearn import preprocessing

scaled_data = preprocessing.scale(X)

pca = PCA(n_components=2) # create a PCA object
pca.fit(scaled_data) # do the math
X_pca = pca.transform(scaled_data) # get PCA coordinates for scaled_data

print(X_pca.shape)
print(y.shape)
```

```
(100, 2)
(100,)
```

Then, we re-build the Perceptron algorithm with the new 2-dimensional dataset. This time, we have only 2 weights w1,w2 and 1 bias b to update.

```python
def h(pc1, pc2, w1, w2, b):
    return np.sign(pc1*w1 + pc2*w2 + b)

w1 = 0
w2 = 0
b = 0
alpha = 1

for j in range(1000):
  for i in range(0, len(X_pca)):
    if h(X_pca[i][0], X_pca[i][1], w1, w2, b) != y[i]:
      w1 = w1 + alpha*y[i]*X_pca[i][0]
      w2 = w2 + alpha*y[i]*X_pca[i][1]
      b = b + alpha*y[i]

print(w1)
print(w2)
print(b)
```

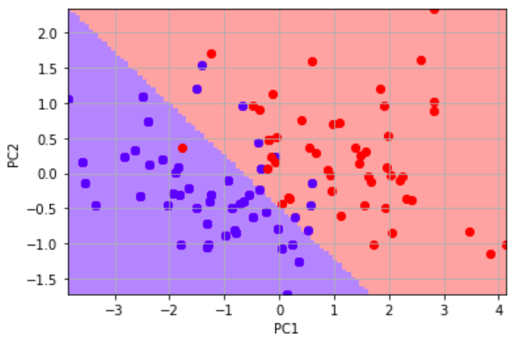
```
2.873257795532451
3.899221277224042
2
```

```python
def which_flower(pc1, pc2):
    return np.sign(pc1*w1 + pc2*w2 + b)

which_flower(2,3)
```

```
1.0
```

```python
#MESH
from matplotlib.colors import ListedColormap
cmap_light = ListedColormap(['#AAAAFF','#AAFFAA','#FFAAAA'])
#generate all the points in the plane using np.meshgrid(np.arange(x_min, x_min, super_small))
xx, yy = np.meshgrid(np.linspace(X_pca[:,0].min(), X_pca[:,0].max(), num=100), np.linspace(X_pca[:,1].min(), X_pca[:,1].max(), num=100))
Z = which_flower(xx.ravel(),yy.ravel())
Z = Z.reshape(xx.shape)

plt.figure()
# xx, yy are ALL the points in the plane, Z is the color of each area
# then plot the points using plt.scatter as usual
plt.pcolormesh(xx,yy,Z,cmap=cmap_light)

# Graph points
for i in range(0, len(X_pca)):
    if y[i] == 1:
        plt.scatter(X_pca[i][0], X_pca[i][1], c = 'r')
    elif y[i] == -1:
        plt.scatter(X_pca[i][0], X_pca[i][1], c = 'b')
plt.grid()
plt.xlabel('PC1')
plt.ylabel('PC2')

plt.xlim(xx.min(),xx.max())
plt.ylim(yy.min(),yy.max())
#------x_min x_max y_min y_max
```

```
(-1.7161812507853758, 2.3299089668338917)
```



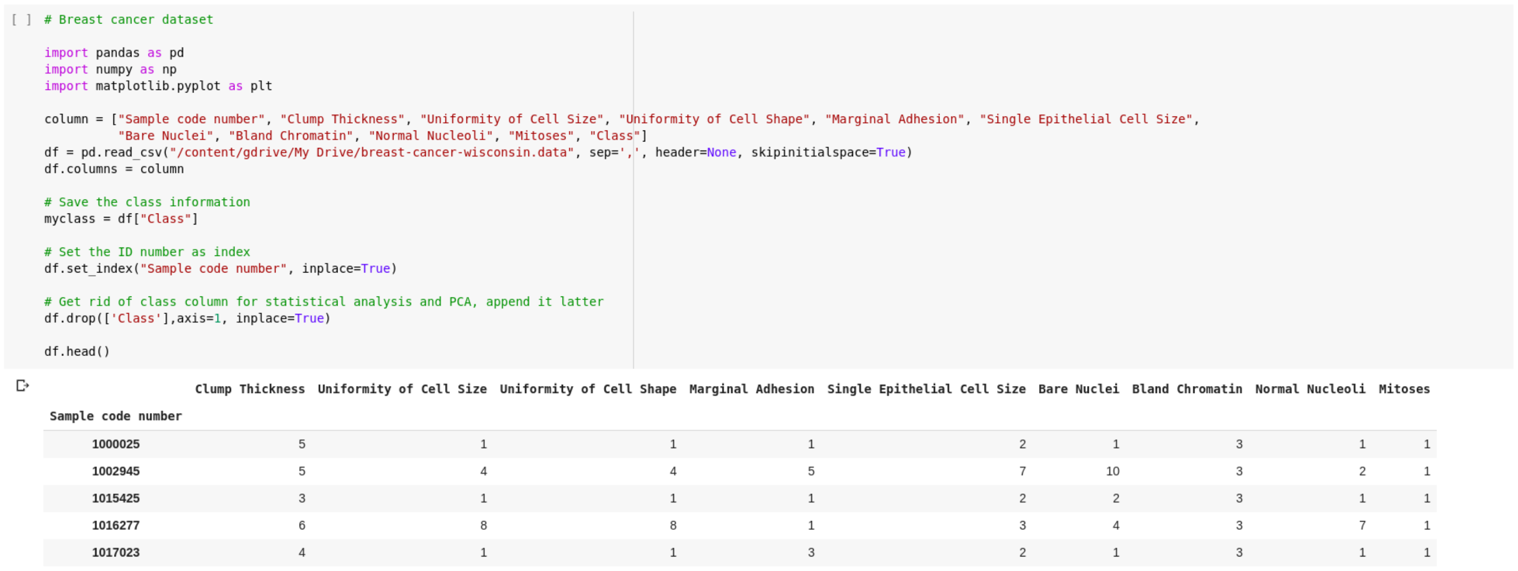
We illustrate the final result with mesh color plot using matplotlib.pyplot. It is obvious that the hyperplane do a great job in dividing 2 distinct classes.

<u>Comment on the convergence rate of Perceptron on Iris:</u>

- The convergence rate of the dataset relies on our initialized weights' values, number of iterations and most importantly, the learning rate alpha.

- For this particular dataset, we would say that it converges relatively fast. Given alpha=1, all the weights and bias is initially set at 0, the algorithm gives 93% accurate predictions after just 100 iterations. And as we continue to experiment, our algorithm reaches 97% accurate predictions at 1000 iterations.

For further study, we implement Perceptron with Breast Cancer dataset. But this time, we use the built-in sklearn's perceptron for convenience.



```
[ ] # Breast cancer dataset

    import pandas as pd
    import numpy as np
    import matplotlib.pyplot as plt

    column = ["Sample code number", "Clump Thickness", "Uniformity of Cell Size", "Uniformity of Cell Shape", "Marginal Adhesion", "Single Epithelial Cell Size",
              "Bare Nuclei", "Bland Chromatin", "Normal Nucleoli", "Mitoses", "Class"]
    df = pd.read_csv("/content/gdrive/My Drive/breast-cancer-wisconsin.data", sep=',', header=None, skipinitialspace=True)
    df.columns = column

    # Save the class information
    myclass = df["Class"]

    # Set the ID number as index
    df.set_index("Sample code number", inplace=True)

    # Get rid of class column for statistical analysis and PCA, append it latter
    df.drop(['Class'],axis=1, inplace=True)

    df.head()
```

	Clump Thickness	Uniformity of Cell Size	Uniformity of Cell Shape	Marginal Adhesion	Single Epithelial Cell Size	Bare Nuclei	Bland Chromatin	Normal Nucleoli	Mitoses
Sample code number									
1000025	5	1	1	1	2	1	3	1	1
1002945	5	4	4	5	7	10	3	2	1
1015425	3	1	1	1	2	2	3	1	1
1016277	6	8	8	1	3	4	3	7	1
1017023	4	1	1	3	2	1	3	1	1

This dataset has 2 classes (benign or malignant) given 9 different attributes ranging from 1 to 10.

We again prepare our X (data) and y (label), divide them into train/test set with the ratio of 90:10.

Then, we import the Perceptron from sklearn.linear_model. The perceptron.coef_ gives all the updated weights.

We use to model to make our predictions on the test set and calculate the accuracy score. 98.57% is a really good score for prediction.

```
[ ] X = df.replace("?", 1).values
    y = myclass.values

    from sklearn.model_selection import train_test_split
    X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=0.1,random_state=4)
```

```
[ ] from sklearn.linear_model import Perceptron

    perceptron = Perceptron(tol=1e-3, random_state=0)
    perceptron.fit(X_train, y_train)

    # The obtained weights after running the algorithm
    print(perceptron.coef_)
```

```
[[-4. 19.  6.  4. -2. 15.  5.  6. 11.]]
```

```
[ ] y_pred = perceptron.predict(X_test)
    print(y_pred)
```

```
[4 2 2 2 2 2 2 4 4 4 2 2 2 2 4 4 2 2 2 2 2 4 2 2 2 4 2 2 4 4 2 4 2 2 2 4 2
 4 4 2 2 4 4 2 2 2 2 2 2 2 2 4 2 2 2 2 4 4 4 4 2 2 4 4 4 2 2 2 4 2]
```

```
[ ] sm.accuracy_score(y_pred, y_test)
```

```
0.9857142857142858
```

II) K Nearest Neighbors.

2.1 Iris dataset

```
[ ] import numpy as np
    import pandas as pd
    import matplotlib.pyplot as plt

    column = ["sepal length", "sepal width", "petal length", "petal width", "class"]
    df = pd.read_csv("/content/gdrive/My Drive/iris.data", sep=',', header=None)
    df.columns = column
    myclass = df["class"]
    df.head(5)
```

	sepal length	sepal width	petal length	petal width	class
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

```
[ ] X = df.drop(["class"], axis=1).values
    y = myclass.values
```

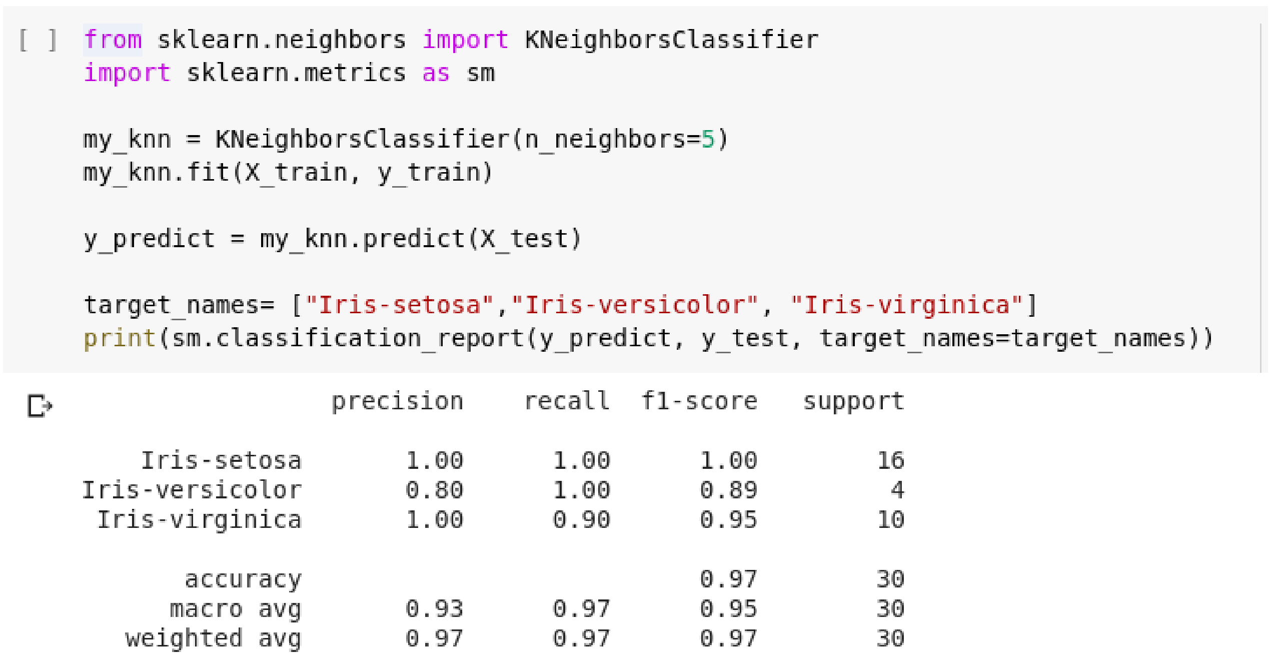
```
[ ] # splitting the data into training and test sets (80:20)
    from sklearn.model_selection import train_test_split
    X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=0.2,random_state=4)
```

```
[ ] print(len(X_train))
    print(len(y_train))
```

```
120
120
```

KNN algorithm can work on multiple classes, not binary like Perceptron. Therefore, we quickly prepare our data: read it into dataframe with pandas, divide train/test (ratio 80:20).

We first implement to KNN with k=5, then report all the score of the dataset by comparing the prediction with the test labels.



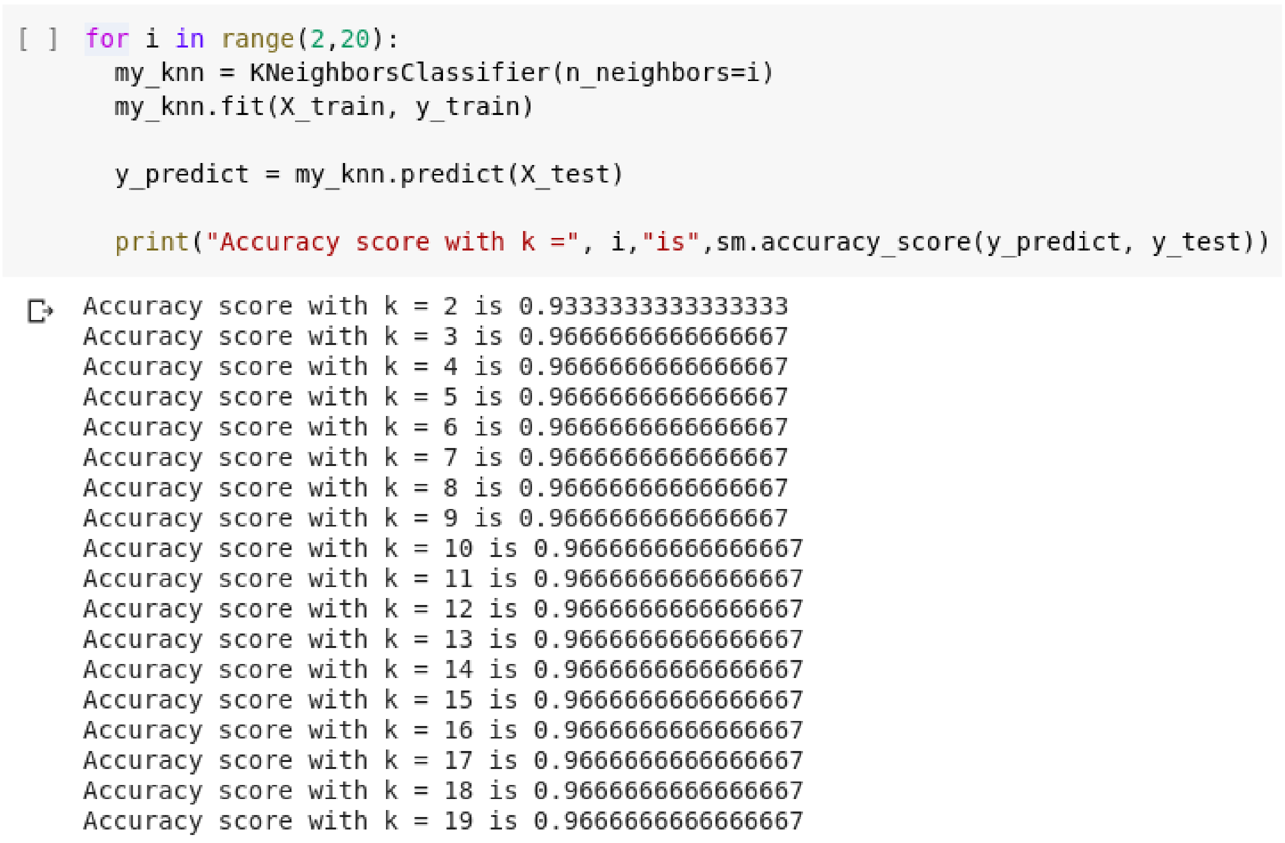
```
[ ] from sklearn.neighbors import KNeighborsClassifier
    import sklearn.metrics as sm

    my_knn = KNeighborsClassifier(n_neighbors=5)
    my_knn.fit(X_train, y_train)

    y_predict = my_knn.predict(X_test)

    target_names= ["Iris-setosa","Iris-versicolor", "Iris-virginica"]
    print(sm.classification_report(y_predict, y_test, target_names=target_names))
```

```
                 precision    recall  f1-score   support

    Iris-setosa       1.00      1.00      1.00        16
Iris-versicolor       0.80      1.00      0.89         4
 Iris-virginica       1.00      0.90      0.95        10

       accuracy                           0.97        30
      macro avg       0.93      0.97      0.95        30
   weighted avg       0.97      0.97      0.97        30
```

It is clear that our model with k=5 works well in predicting all 3 classes. The overall accuracy is up to 97%, leaving only 3% classification error.

Then we vary k value from 2 to 19, then calculate the accuracy of each KNN.

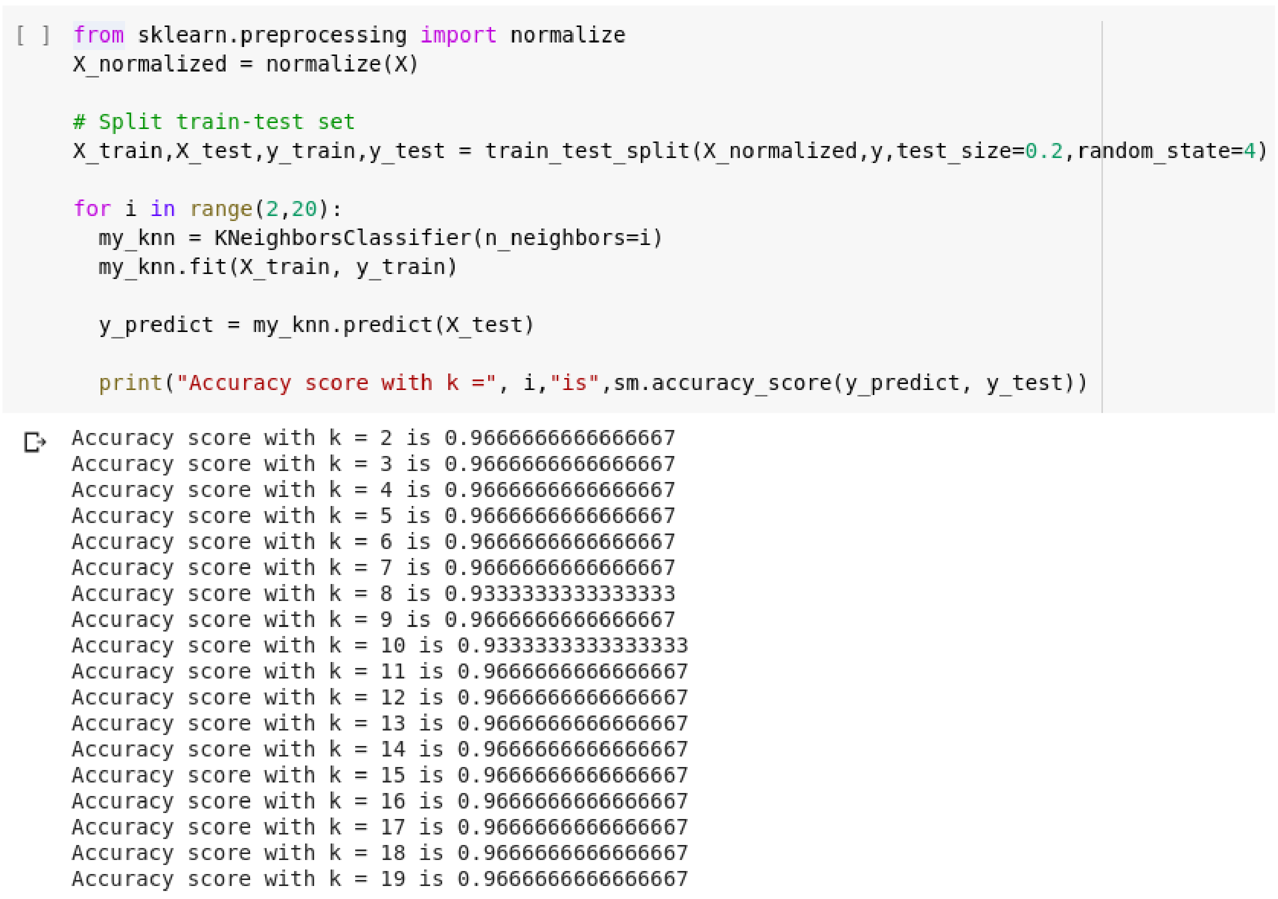
The accuracy score remains constant when varying k.



```
[ ]  for i in range(2,20):
       my_knn = KNeighborsClassifier(n_neighbors=i)
       my_knn.fit(X_train, y_train)

       y_predict = my_knn.predict(X_test)

       print("Accuracy score with k =", i,"is",sm.accuracy_score(y_predict, y_test))
```

```
Accuracy score with k = 2 is 0.9333333333333333
Accuracy score with k = 3 is 0.9666666666666667
Accuracy score with k = 4 is 0.9666666666666667
Accuracy score with k = 5 is 0.9666666666666667
Accuracy score with k = 6 is 0.9666666666666667
Accuracy score with k = 7 is 0.9666666666666667
Accuracy score with k = 8 is 0.9666666666666667
Accuracy score with k = 9 is 0.9666666666666667
Accuracy score with k = 10 is 0.9666666666666667
Accuracy score with k = 11 is 0.9666666666666667
Accuracy score with k = 12 is 0.9666666666666667
Accuracy score with k = 13 is 0.9666666666666667
Accuracy score with k = 14 is 0.9666666666666667
Accuracy score with k = 15 is 0.9666666666666667
Accuracy score with k = 16 is 0.9666666666666667
Accuracy score with k = 17 is 0.9666666666666667
Accuracy score with k = 18 is 0.9666666666666667
Accuracy score with k = 19 is 0.9666666666666667
```

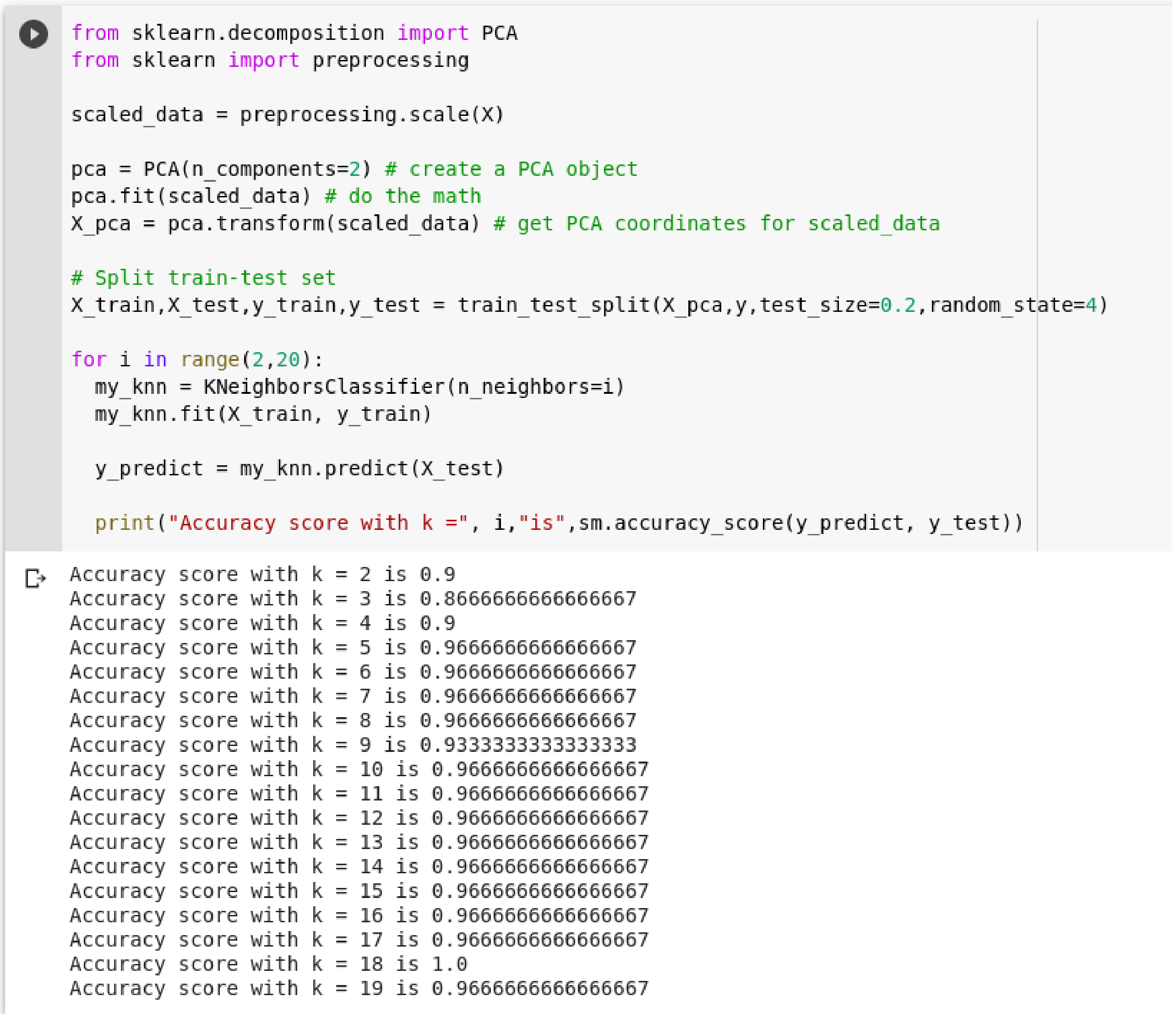
Then, we normalize the dataset and repeat the same process.
The result comes out nearly the same as applying KNN directly.



```
[ ]  from sklearn.preprocessing import normalize
     X_normalized = normalize(X)

     # Split train-test set
     X_train,X_test,y_train,y_test = train_test_split(X_normalized,y,test_size=0.2,random_state=4)

     for i in range(2,20):
       my_knn = KNeighborsClassifier(n_neighbors=i)
       my_knn.fit(X_train, y_train)

       y_predict = my_knn.predict(X_test)

       print("Accuracy score with k =", i,"is",sm.accuracy_score(y_predict, y_test))
```

```
Accuracy score with k = 2 is 0.9666666666666667
Accuracy score with k = 3 is 0.9666666666666667
Accuracy score with k = 4 is 0.9666666666666667
Accuracy score with k = 5 is 0.9666666666666667
Accuracy score with k = 6 is 0.9666666666666667
Accuracy score with k = 7 is 0.9666666666666667
Accuracy score with k = 8 is 0.9333333333333333
Accuracy score with k = 9 is 0.9666666666666667
Accuracy score with k = 10 is 0.9333333333333333
Accuracy score with k = 11 is 0.9666666666666667
Accuracy score with k = 12 is 0.9666666666666667
Accuracy score with k = 13 is 0.9666666666666667
Accuracy score with k = 14 is 0.9666666666666667
Accuracy score with k = 15 is 0.9666666666666667
Accuracy score with k = 16 is 0.9666666666666667
Accuracy score with k = 17 is 0.9666666666666667
Accuracy score with k = 18 is 0.9666666666666667
Accuracy score with k = 19 is 0.9666666666666667
```

Afterwards, we apply PCA before implementing the KNN algorithm.

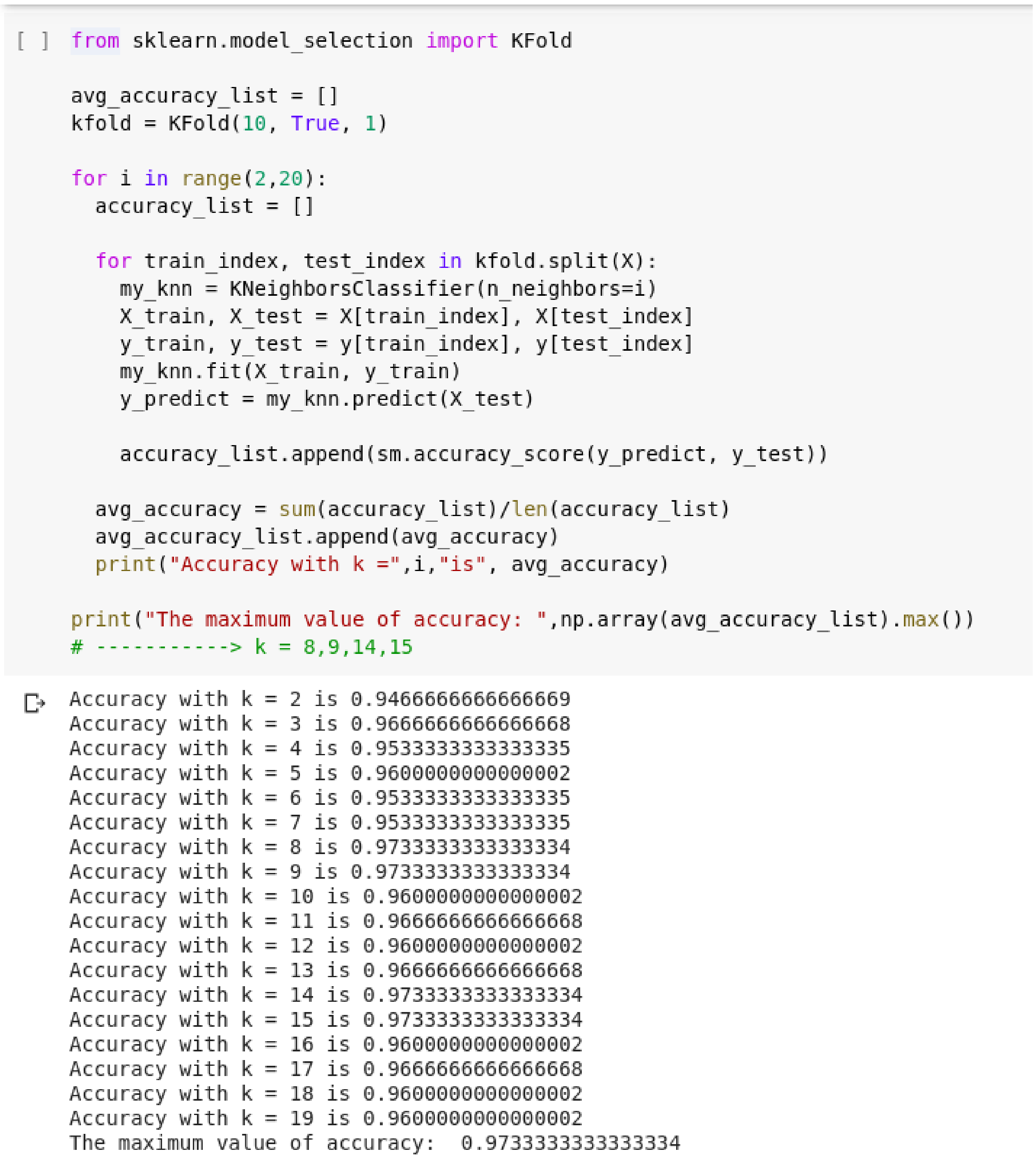


```python
from sklearn.decomposition import PCA
from sklearn import preprocessing

scaled_data = preprocessing.scale(X)

pca = PCA(n_components=2) # create a PCA object
pca.fit(scaled_data) # do the math
X_pca = pca.transform(scaled_data) # get PCA coordinates for scaled_data

# Split train-test set
X_train,X_test,y_train,y_test = train_test_split(X_pca,y,test_size=0.2,random_state=4)

for i in range(2,20):
  my_knn = KNeighborsClassifier(n_neighbors=i)
  my_knn.fit(X_train, y_train)

  y_predict = my_knn.predict(X_test)

  print("Accuracy score with k =", i,"is",sm.accuracy_score(y_predict, y_test))
```

```
Accuracy score with k = 2 is 0.9
Accuracy score with k = 3 is 0.8666666666666667
Accuracy score with k = 4 is 0.9
Accuracy score with k = 5 is 0.9666666666666667
Accuracy score with k = 6 is 0.9666666666666667
Accuracy score with k = 7 is 0.9666666666666667
Accuracy score with k = 8 is 0.9666666666666667
Accuracy score with k = 9 is 0.9333333333333333
Accuracy score with k = 10 is 0.9666666666666667
Accuracy score with k = 11 is 0.9666666666666667
Accuracy score with k = 12 is 0.9666666666666667
Accuracy score with k = 13 is 0.9666666666666667
Accuracy score with k = 14 is 0.9666666666666667
Accuracy score with k = 15 is 0.9666666666666667
Accuracy score with k = 16 is 0.9666666666666667
Accuracy score with k = 17 is 0.9666666666666667
Accuracy score with k = 18 is 1.0
Accuracy score with k = 19 is 0.9666666666666667
```

The results is worse compared to previous approaches, due to the fact that PCA creates information loss to the original dataset.

Our KNN algorithm is based mainly on the value k chosen. So to improve our model, we propose a method of using k-cross validation to find the average accuracy after multiple implementation on different folds.

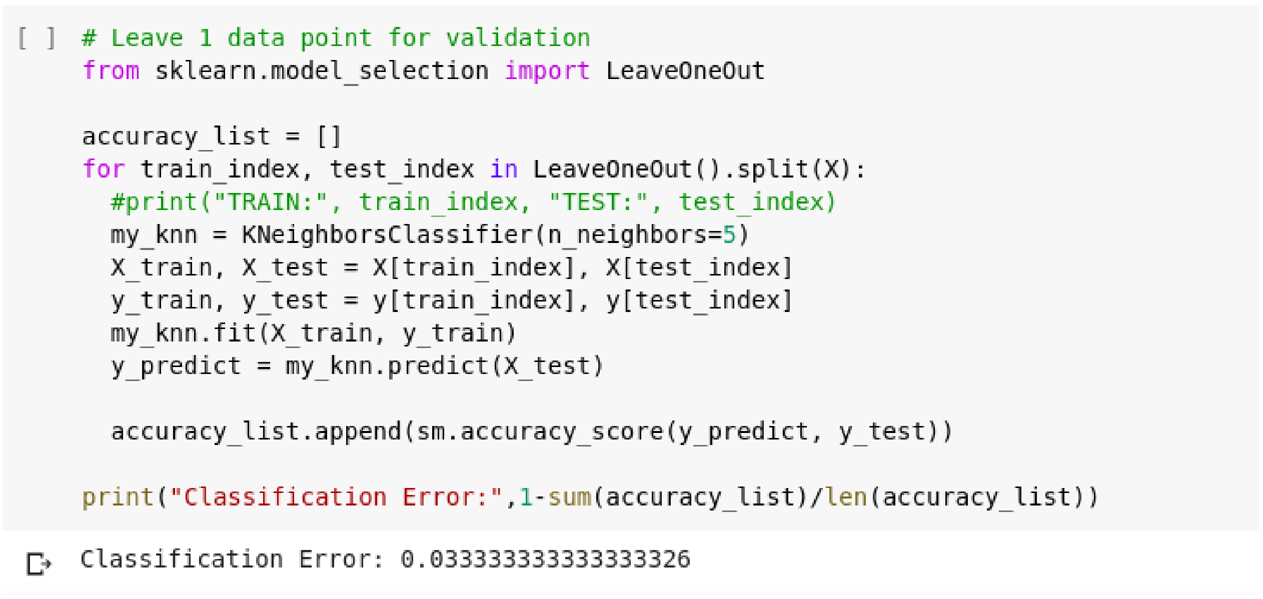


```
[ ] from sklearn.model_selection import KFold

    avg_accuracy_list = []
    kfold = KFold(10, True, 1)

    for i in range(2,20):
      accuracy_list = []

      for train_index, test_index in kfold.split(X):
        my_knn = KNeighborsClassifier(n_neighbors=i)
        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = y[train_index], y[test_index]
        my_knn.fit(X_train, y_train)
        y_predict = my_knn.predict(X_test)

        accuracy_list.append(sm.accuracy_score(y_predict, y_test))

      avg_accuracy = sum(accuracy_list)/len(accuracy_list)
      avg_accuracy_list.append(avg_accuracy)
      print("Accuracy with k =",i,"is", avg_accuracy)

    print("The maximum value of accuracy: ",np.array(avg_accuracy_list).max())
    # -----------> k = 8,9,14,15
```

```
Accuracy with k = 2 is 0.9466666666666669
Accuracy with k = 3 is 0.9666666666666668
Accuracy with k = 4 is 0.9533333333333335
Accuracy with k = 5 is 0.9600000000000002
Accuracy with k = 6 is 0.9533333333333335
Accuracy with k = 7 is 0.9533333333333335
Accuracy with k = 8 is 0.9733333333333334
Accuracy with k = 9 is 0.9733333333333334
Accuracy with k = 10 is 0.9600000000000002
Accuracy with k = 11 is 0.9666666666666668
Accuracy with k = 12 is 0.9600000000000002
Accuracy with k = 13 is 0.9666666666666668
Accuracy with k = 14 is 0.9733333333333334
Accuracy with k = 15 is 0.9733333333333334
Accuracy with k = 16 is 0.9600000000000002
Accuracy with k = 17 is 0.9666666666666668
Accuracy with k = 18 is 0.9600000000000002
Accuracy with k = 19 is 0.9600000000000002
The maximum value of accuracy:  0.9733333333333334
```

We divide the dataset into 10 folds, record the accuracy score for each fold then averaging them. After that, we compare all the obtained averaged accuracy score. It is evident that k = 8,9,14,15 are the best values for k in our KNN algorithm.

Finally, we apply the Leave One Out – a special case of k-cross validation where number of folds = number of data points, leaving only 1 point for validation.



```
# Leave 1 data point for validation
from sklearn.model_selection import LeaveOneOut

accuracy_list = []
for train_index, test_index in LeaveOneOut().split(X):
  #print("TRAIN:", train_index, "TEST:", test_index)
  my_knn = KNeighborsClassifier(n_neighbors=5)
  X_train, X_test = X[train_index], X[test_index]
  y_train, y_test = y[train_index], y[test_index]
  my_knn.fit(X_train, y_train)
  y_predict = my_knn.predict(X_test)

  accuracy_list.append(sm.accuracy_score(y_predict, y_test))

print("Classification Error:",1-sum(accuracy_list)/len(accuracy_list))
```

```
Classification Error: 0.033333333333333326
```

Classification Error is 3.33% as calculated.