STATISTICAL LEARNING AND NEURAL NETWORKS, A.A. 2022/2023

COMPUTER LAB 1 - k-NN classifier

Duration: 6 hours

Exercise 1 - Synthetic dataset

In this exercise, you will employ a synthetic dataset (file Lab1_Ex_1_Synthtetic.hdf5), containing labelled training data and test data for two classes. For each example the first two columns represent the features, while the last column represents the label.

Task: your task is to implement a k-NN classifier, which calculates the probability that a given test example belongs to each class, and outputs a class label as the class with the highest probability. You will evaluate the classifier performance computing the average classification accuracy (i.e. the fraction of test examples that have been classified correctly in respect to the full test set).

In particular, you should perform the following:

- Train a k-NN classifier for different values of k.
- Compare accuracy on the training set and the test set. Calculating accuracy of the training set means that you will have to classify each sample in the training set as if it were a test sample; one expects that classification of training samples will perform well, and this may also be used to validate your implementation. Accuracy is defined as the ratio between the number of test samples that are correctly classified, and the total number of test samples. Create a graph using the matplotlib library showing the evolution of the accuracy for different values of k over the test set. Create a second graph to show the evolution of the accuracy for different values of k over the train set and compare the two.
- Identifying overfitting and underfitting in the obtained results.

Note that, for this computer lab, you do not need to employ a validation set.

Other indications:

- The student is required to implement the k-NN algorithm from scratch. Only the numpy library is allowed, while other libraries such as scikit_learn are forbidden.

```
 1 import numpy as np
 2 import matplotlib.pyplot as plt
 3 import h5py
 4 import math
 5 import pandas as pd
 6
 7 #Implement a function to compute the Euclidean distance between two vectors, and one to implemer
 8 #   - Taking a sample
 9 #   - Computing all the distances between the sample element and the elements of the training se
10 #   - sort the the training set based on the distances to the element (the use of functions like
11 #   - select the top k elements in terms of distance
12 #   - evaluate to which class the majority of these k elements belongs to (e.g., it is possible
```

```
#Change the path to match the position of your file
#The Dataset can be loaded using thhe file option in Google Colab (the directory icon on the le
Dataset1 = h5py.File('/content/Lab1_Ex_1_Synthtetic.hdf5')
Data = np.array(Dataset1.get('Dataset'))


Data.ndim

Train_Set = Data[:200,:]
Test_Set = Data[200:,:]

#To be completed by the student
#Function to calculate Euclideandistance

def eucdist (sample, data):    #   - Taking a sample
    dist = np.sqrt((sample[0]-data[0])**2 + (sample[1]-data[1])**2)
    return dist


#Defining K-NN

def k_nn (sample,n):
    knn_mat = Train_Set
    dist=[]
    for j in range (len(knn_mat)):
         dist.append(eucdist(sample,knn_mat[j]))

    knn_mat = pd.DataFrame(knn_mat)
    knn_mat.insert(loc=3, column=3, value=dist)
    knn_mat = knn_mat.sort_values(3)
    knn = knn_mat.values[:n]

    class1 = 0
    class2 = 0

    for k in range(len(knn)):
       if knn[k,2]  == 0 :
        class1 += 1
       else:
        class2 += 1


    if class1>class2 :
        return 0
    else :
        return 1


k = 10
a = k_nn(Test_Set[199] ,k)
print('The class is : ' , a)

    The class is :  1
```

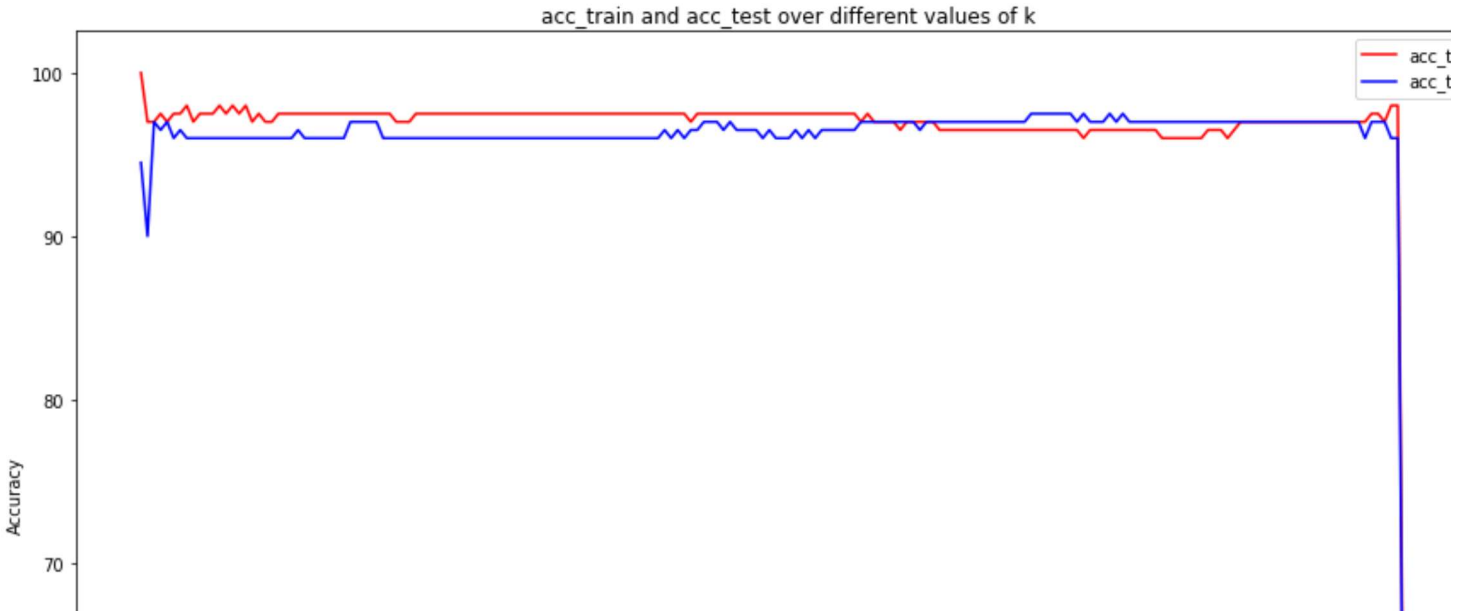
So far we have created the KNN classifier, No we calculate the Accuracy

```
def acc (test , k):
    true = 0
    test_predicted = 0
    for i in range(len(test)):
        test_predicted = (k_nn(test[i] ,k))
        if test [i][2] == test_predicted:
            true += 1

    return (true / len(test ))* 100

acc_test =  acc(Test_Set, 10)
print('The Accuracy for the test dataset is:', acc_test)
acc_train =  acc(Train_Set, 10)
print('The Accuracy for the train dataset is:',acc_train)

```

```
The Accuracy for the test dataset is: 96.0
The Accuracy for the train dataset is: 97.5
```

By calculating the accuracy we can see that the the training data set has a bit higher accuracy and hence there's no overfitting . Now , we plot the accuracy for different values of k

```
k = []
acc_train_mat = []
acc_test_mat = []
for i in range(1,len(Train_Set)):
    k.append(i)
    acc_train_mat.append(acc(Train_Set , i))
    acc_test_mat.append(acc(Test_Set , i))

f = plt.figure()
f.set_figwidth(15)
f.set_figheight(10)
plt.plot(k, acc_train_mat , color = 'red')
plt.plot(k, acc_test_mat , color = 'blue')
plt.legend(['acc_train', 'acc_test'])
plt.title('acc_train and acc_test over different values of k')
plt.xlabel ('k')
plt.ylabel('Accuracy')
plt.show()
```



Student's comments to exercise 1

The k-nearest neighbors algorithm, also known as KNN or k-NN, is a non-parametric, supervised (Train data have label) learning classifier, which uses proximity (We choose euc distance) to make classifications or predictions about the grouping of an individual data point. KNN sometimes called a lazy learner algorithm as it just store data rather than learning from data for optimizing some parameter.

1. The k value in the k-NN algorithm defines how many neighbors will be checked to determine the classification of a specific query point.
2. It is therefore very intensive to the value of k and as k increases the accuracy is decreasing.(The initial increase is for k = 0 which is not important)
3. The figure depicts best results archieved with the values of k < 120. In this period there is no overfitting and the accuracy is acceptable and above 90% for both of the training and testing data set. with k increasing , the underfitting happens as acc_test > acc_train . and the worst scenario happens near k = 200 with around 50% accuracy wich is almost terrible for a binary classification problem.

Overall speaking, One the one hand, a small value of k, means that noise will have a higher influence on the result. On the other hand, if k is selected to be too large, the model becomes too generalized and fails to accurately predict the data points in both train and test sets.

Exercise 2 - Wine dataset

Part 1

In this exercise, a real problem will be examined. The dataset used in this exercise was derived from wine quality dataset from the work "*Modeling wine preferences by data mining from physicochemical properties*" by P. Cortez, A. Cerdeira, F. Almeida, T. Matos and J. Reis.

For each element of the dataset 11 features are provided, representing different wine characteristics, such as density, pH and alcholic content, and the final column consists of a quality evaluation on a scale from 1 to 10. More information can be found at https://archive.ics.uci.edu/ml/datasets/wine+quality.

A subset of the dataset containing 400 elements is provided. Create a training set and a test set of 200 samples each. The objective is to:

- Predict the wine quality over the test set using the k-NN algorithm and evaluating the prediction accuracy for different values of k. Create a graph using the matplotlib library showing the evolution of the accuracy for different values of k over the test set.
- Identifying overfitting and underfitting in the obtained results.

Part 2

The prediction of the wine quality could also be framed as a regression. Estimate the accuracy and the Mean Square Error achieved using linear resgression. For this task is possible to use the library sklearn and the function linear_model.LinearRegression()

```
#Defining Eucdist calculator function

Dataset2 = h5py.File('/content/Lab1_Ex_2_wine.hdf5')

Data = np.array(Dataset2.get('Dataset'))

# Normalizing data
Train_Set = Data[:200,:]
Train_Set = pd.DataFrame(Train_Set)
y = Train_Set[11]
Train_Set = Train_Set.drop(columns=11)
Train_Set=(Train_Set-Train_Set.mean())/Train_Set.std()
Train_Set.insert(loc=11, column=11 ,value= y)
Train_Set = Train_Set.values

# Normalizing data
Test_Set = Data[200:,:]
Test_Set = pd.DataFrame(Test_Set)
y = Test_Set[11]
Test_Set = Test_Set.drop(columns=11)
Test_Set = (Test_Set-Test_Set.mean())/Test_Set.std()
Test_Set.insert(loc=11, column=11 ,value= y)
Test_Set = Test_Set.values


def eucdist (sample, data):    #   - Taking a sample
    dist = 0
    for i in range(len(data)-1):
        dist += (sample[i]-data[i])**2
    return np.sqrt(dist)

# Defining knn

def k_nn (sample,k):
    knn_mat = Train_Set
    dist=[]
    for j in range (len(knn_mat)):
         dist.append(eucdist(sample,knn_mat[j]))

    knn_mat = pd.DataFrame(knn_mat)
    knn_mat.insert(loc=12, column=12, value=dist)
    knn_mat = knn_mat.sort_values(12)
    knn = knn_mat.values[:k]

# class prediction section

```

```
    classmat = np.array(range(10)) + 1
    predictmat = np.zeros(10)

    for k in range(len(knn)):
       for j in classmat:
        if knn[k,11]  == j :
            predictmat[j-1] += 1

    return np.argmax(predictmat) + 1


a = k_nn(Test_Set[12] ,10)
print(a)
```

```
    6
```

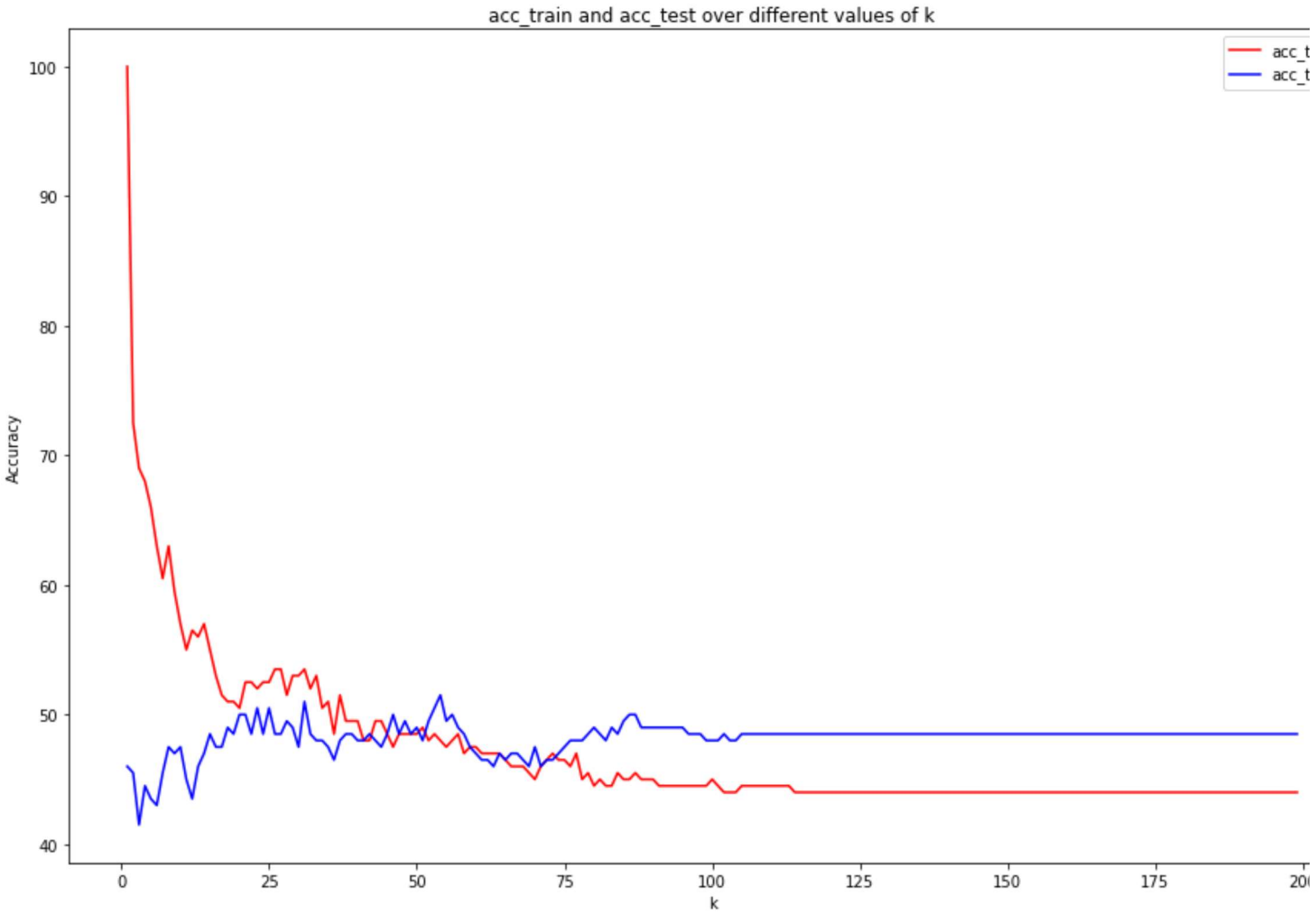
Double-click (or enter) to edit

```
# Calculating Acc
def acc (test , k):
    true = 0
    test_predicted = 0
    for i in range(len(test)):
        test_predicted = (k_nn(test[i] ,k))
        if test [i][11] == test_predicted:
            true += 1

    return (true / len(test ))* 100

acc_test =  acc(Test_Set, 5)
print(acc_test)
acc_train =  acc(Train_Set, 5)
print(acc_train)
```

```
    43.5
    66.0
```

```
k = []
acc_train_mat = []
acc_test_mat = []
for i in range(1,len(Train_Set)):
    k.append(i)
    acc_train_mat.append(acc(Train_Set , i))
    acc_test_mat.append(acc(Test_Set , i))

f = plt.figure()
f.set_figwidth(15)
f.set_figheight(10)
plt.plot(k, acc_train_mat , color = 'red')
plt.plot(k, acc_test_mat , color = 'blue')
plt.legend(['acc_train', 'acc_test'])
```

```
plt.title('acc_train and acc_test over different values of k')
plt.xlabel ('k')
plt.ylabel('Accuracy')
plt.show()
```




```
#Part 2
Dataset2 = h5py.File('/content/Lab1_Ex_2_wine.hdf5')
Data = np.array(Dataset2.get('Dataset'))

# Normalizing and saving train data labels

Train_Set = Data[:200,:]
Train_Set = pd.DataFrame(Train_Set)
y_train = Train_Set[11]
y_train = np.array(y_train.values)
Train_Set = Train_Set.drop(columns=11)
Train_Set=(Train_Set-Train_Set.mean())/Train_Set.std()
Train_Set = np.array(Train_Set.values)

# Normalizing and saving test data labels

Test_Set = Data[200:,:]
Test_Set = pd.DataFrame(Test_Set)
y_test = Test_Set[11]
y_test = np.array(y_test.values)
```

```
Test_Set = Test_Set.drop(columns=11)
Test_Set = (Test_Set - Test_Set.mean()) / Test_Set.std()
Test_Set = np.array(Test_Set.values)

from sklearn import linear_model
clf = linear_model.LinearRegression()

# Training the model
clf.fit(Train_Set , y_train)
Test_predicted = np.rint(clf.predict(Test_Set))
Train_predicted = np.rint(clf.predict(Train_Set))

#To be completed by the student
```

```
# Calculating Acc
def accuracy (test , predict):
    true = 0
    for i in range(len(test)):
        if test[i] == predict[i]:
            true += 1

    return (true / len(test))* 100

acc_test =  accuracy(Test_predicted, y_test)
print(acc_test)
acc_train =  accuracy(Train_predicted, y_train)
print(acc_train)
```

```
45.0
47.0
```

Student's comments to exercise 2

- KNN

1. We normalize data in order to avoid KNN assigning higher weights to the features with a higher amplitude.
2. The k value in the k-NN algorithm defines how many neighbors will be checked to determine the classification of a specific query point an It is therefore very intensive to the value of k and as k increases the accuracy is decreasing.(The initial increase is for k = 0 which is not important)
3. The figure show that there's a huge overfitting for the k < 25 while underfitting happens for most of the values of k>50. The total accuracy is not acceptable. The reason probably is Unbalanced class distribution, meaning that some classes are underrepresented, hence KNN suffering.

- Linear Regression

1. We basically use Linear regression to regress some continous parameter based on a series of other parameters; hence, linear regression(and desecretizing its result using rounding) is not the best solution for classification(descrete).

Exercise 3: Phoneme Dataset

In this exercise the Phoneme dataset is examined https://catalog.ldc.upenn.edu/LDC93s1. Each line represents 256 samples gathered at a 16 kHz of different speech signals. The objective is to classify wether the sound emitted is a "sh", "iy", "dcl", "aa", "ao" phoneme.

Again, a subset of the dataset containing 400 elements is provided. Create a training set and a test set of 200 samples each.

- Classify the samples which compose the test set using the k-NN algorithm and evaluate the prediction accuracy for different values of k. Create a graph using the matplotlib library showing the evolution of the accuracy for different values of k over the test set.
- Identifying overfitting and underfitting in the obtained results.

```
Dataset3 = h5py.File('/content/Lab1_Ex_3.hdf5')

Data = np.array(Dataset3.get('Dataset'))

# Normalizing data
Train_Set = Data[:200,:]

Train_Set = pd.DataFrame(Train_Set)
y = Train_Set[256]
Train_Set = Train_Set.drop(columns=256)
Train_Set=(Train_Set-Train_Set.mean())/Train_Set.std()
Train_Set.insert(loc=256, column=256 ,value= y)
Train_Set = Train_Set.values

# Normalizing data
Test_Set = Data[200:,:]
Test_Set = pd.DataFrame(Test_Set)
y = Test_Set[256]
Test_Set = Test_Set.drop(columns=256)
Test_Set = (Test_Set-Test_Set.mean())/Test_Set.std()
Test_Set.insert(loc=256, column=256 ,value= y)
Test_Set = Test_Set.values


# Ecuc dist

def eucdist (sample, data):    #   - Taking a sample
    dist = 0
    for i in range(len(data)-1):
        dist += (sample[i]-data[i])**2
    return np.sqrt(dist)

# Defining knn

def k_nn (sample,k):
    knn_mat = Train_Set
    dist=[]
    for j in range (len(knn_mat)):
         dist.append(eucdist(sample,knn_mat[j]))

    knn_mat = pd.DataFrame(knn_mat)
    knn_mat.insert(loc=257, column=257, value=dist)
    knn_mat = knn_mat.sort_values(257)
    knn = knn_mat.values[:k]
```

```
45
46 # class prediction section
47
48     classmat = np.array(range(5))
49     predictmat = np.zeros(5)
50
51     for k in range(len(knn)):
52        for j in classmat:
53         if knn[k,256]  == j :
54             predictmat[j] += 1
55
56     return np.argmax(predictmat)
57
58
59
60
61
```

1. We normalize data in order to avoid KNN assigning higher weights to the features with a higher amplitude
2. We can use the same eucdist function as the previous excersise
3. We also can use the same KNN function but with the following changes:

> - Changing the label column (256 instead of 11)
> - Modifying the class prediction section with respect to the number of the classes (5)

```
1 # Calculating Acc
2 def acc (test , k):
3     true = 0
4     test_predicted = 0
5     for i in range(len(test)):
6         test_predicted = (k_nn(test[i] ,k))
7         if test [i][256] == test_predicted:
8             true += 1
9
10    return (true / len(test ))* 100
11
12 acc_test =  acc(Test_Set, 5)
13 print('Test Accuracy:' ,acc_test)
14 acc_train =  acc(Train_Set, 5)
15 print('Train Accuracy:' ,acc_train)
16
```

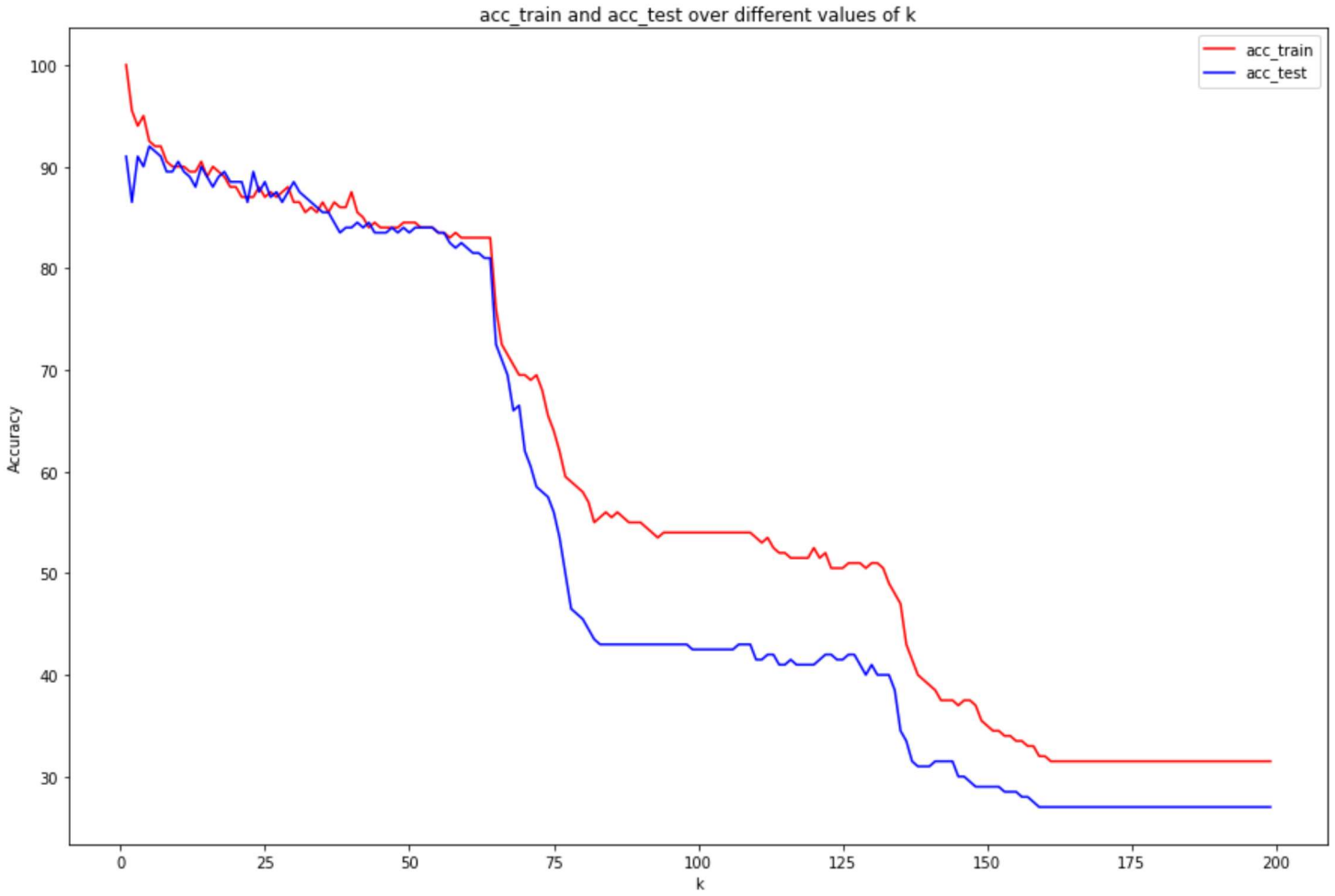
```
Test Accuracy: 92.0
Train Accuracy: 92.5
```

- We measure the accuracy by comparing the KNN prediction with the true label of the data

---

1. The accuracy values are acceptable
2. The small difference between the testing and training datasets' accuracy doesn't demonstrate any overfitting

```
k = []
acc_train_mat = []
acc_test_mat = []
for i in range(1,len(Train_Set)):
    k.append(i)
    acc_train_mat.append(acc(Train_Set , i))
    acc_test_mat.append(acc(Test_Set , i))

f = plt.figure()
f.set_figwidth(15)
f.set_figheight(10)
plt.plot(k, acc_train_mat , color = 'red')
plt.plot(k, acc_test_mat , color = 'blue')
plt.legend(['acc_train', 'acc_test'])
plt.title('acc_train and acc_test over different values of k')
plt.xlabel ('k')
plt.ylabel('Accuracy')
plt.show()
```



Student's comments to exercise 3

1. KNN is a very slow algorithm (Since It must calculate the distance of a sample point from all of the other points in the training dataset) - Specially in cases like this excersise with many features(256), the calculation time can be problematic. (48 minutes execution time for this block)

2. The k value in the k-NN algorithm defines how many neighbors will be checked to determine the classification of a specific query point.
3. It is therefore very intensive to the value of k and as k increases the accuracy is decreasing.
4. One the one hand, a small value of k, means that noise will have a higher influence on the result. On the other hand, if k is selected to be too large, the model becomes too generalized and fails to accurately predict the data points in both train and test sets. This situation is known as underfitting.

STATISTICAL LEARNING AND NEURAL NETWORKS, A.A. 2022/2023

COMPUTER LAB 2 - Model fitting and classification

Duration: 9 hours

Exercise 1 – Model fitting for continuous distributions: Multivariate Gaussian

In this exercise, you will employ a dataset based on the classic dataset *Iris Plants Database* https://archive.ics.uci.edu/ml/datasets/iris. You will be provided a subset of this dataset comprising only two classes (*Iris Setosa* and *Iris Versicolour*), and only two features per class (*petal length* in cm and *petal width* in cm). The objective is to the determine the kind of iris based on the content of the features.

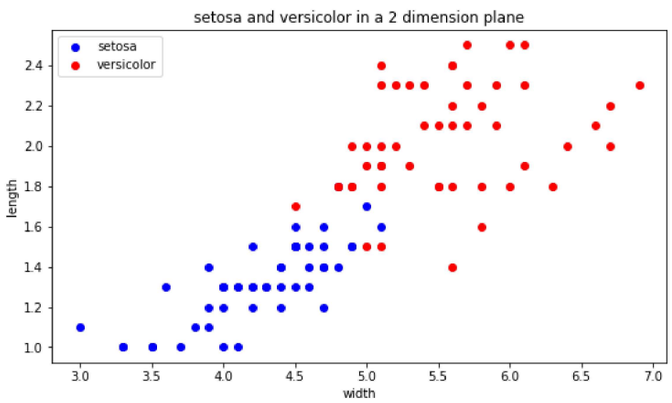
Task: you have to fit class-conditional Gaussian multivariate distributions to the data, and visualize the probability density functions. In particular, you should perform the following:

- Divide the dataset in two parts (*Iris Setosa* which corresponds to class zero and *Iris Versicolour* which correspond to class 1). Then work only on one class at a time.
- Plot the data of each class (use the *plt.scatter( )* function)
- Visualize the histogram of petal length and petal width (use e.g. the *plt.hist( )* function)
- Calculate the maximum likelihood estimate of the mean and covariance matrix under a multivariate Gaussian model, independently for each class (these are the parameters of the class-conditional distributions). Note: is the Gaussian model good for these data?
- Visualize the 2-D joint pdf of petal length and width for the two classes.

```
import numpy as np
import matplotlib.pyplot as plt
import h5py
Dataset1 = h5py.File("/content/Lab2_Ex_1_Iris.hdf5")
data = np.array(Dataset1.get('Dataset'))
```

New section

```
#Separate the dataset in the two classes, you can use the numpy function argsort and unique to do this.
indx0 = np.where(data[:,2] == 0) # Returns a list of indices of the rows which their third element is equal to zero(creates a filter)
indx1 = np.where(data[:,2] == 1) # """""""""""""""""""""""""""""""
caption0 = 'Setosa'
caption1 = 'Versicolour'
setosa = data[indx0][:,0:2]  # uses the index0 list to separate (filter) the rows which their third element is equal to zero and slice the first 2 colu
versicolour = data[indx1][:,0:2] # """""""""""""""""""""""""""""""
#print (20 *'*_',f'{caption0}', 20 *'*_','\n', setosa)
#print (20 *'*_',f'{caption1}', 20 *'*_','\n', versicolour)
#%% Draw the scatter plot of the two classes on the same image
plt.figure(figsize=(9,5))
setosa_petal_length = setosa[:,0]
setosa_petal_width = setosa[:,1]
versicolour_petal_length = versicolour[:,0]
versicolour_petal_width = versicolour[:,1]
plt.scatter(setosa_petal_length, setosa_petal_width , c = 'blue' )
plt.scatter(versicolour_petal_length, versicolour_petal_width, c = 'red')
labels = ['setosa' , 'versicolor']
plt.legend(labels , loc='upper left')
plt.xlabel('width')
plt.ylabel('length')
plt.title("setosa and versicolor in a 2 dimension plane")
plt.show()
#Draw the scatter plot of the two classes on the same image
```




```
#Visualize the histogram of petal length and petal width (use e.g. the plt.hist() function)
plt.figure(figsize=(9,6))
total_petal_length = np.hstack((setosa_petal_length, versicolour_petal_length))
total_petal_width = np.hstack((setosa_petal_width, versicolour_petal_width))
plt.hist(total_petal_length , color= 'brown', histtype='bar', label='length')
plt.hist(total_petal_width, color = 'yellow', histtype='bar', label= 'width')
plt.legend()
plt.xlabel('amplitude')
plt.ylabel('frequenct')
plt.title("petal length and petal width")
plt.show()
```

The figure demostrate sum of two random variables in a gaussian-shape figure

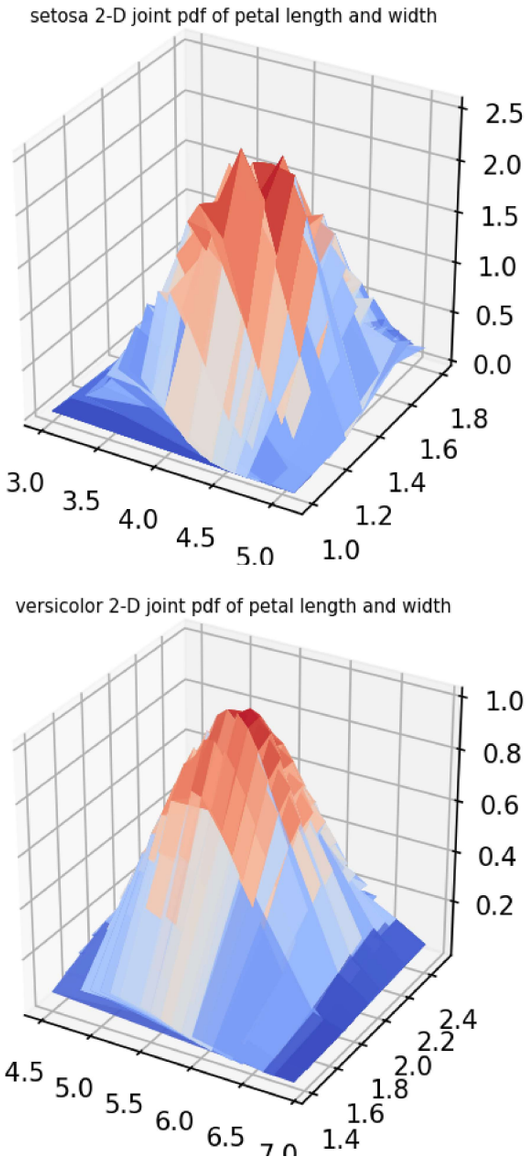
```
#Calculate mean and covariance matrix under a multivariate Gaussian model. Scalar products can be computed with the function np.matmul()

#The transpose can be obtained with the function np.transpose

#To make a scalar product between arrays in the form [Mx1]x[1xM] starting from 1D array A it may be necessary to add a new axis using
#the function A[:,np.newaxis]
def mean_calculator (x):
    x_mean = [0,0]
    for j in range(x.shape[1]):
        for i in range(x.shape[0]):
            x_mean[j] += x[i,j]
        x_mean[j] /= x.shape[0]
    return x_mean
###
def cov_calculator(y):
    y_mean = np.array(mean(y))
    y_cov = np.empty([len(y.shape), len(y.shape)])
    for k in range(y.shape[0]):
        y_cov +=  (np.subtract( y[k][:, np.newaxis] @ y[k][:, np.newaxis].T , y_mean[:, np.newaxis] @ y_mean[:, np.newaxis].T))
    return np.divide(y_cov,y.shape[0])
###
setosa_mean = mean_calculator(setosa)
setosa_cov = cov_calculator(setosa)
versicolour_mean = mean_calculator(versicolour)
versicolour_cov = cov_calculator(versicolour)

print('         MLE estimate:\n\n')

print("* setosa:\n\n",'mean: ',"\n",mean_calculator(setosa),"   ","\n\n",'cov: ',"\n",cov_calculator(setosa))
print (30*'_')
print("\n* versicolour:\n\n",'mean: ',"\n",mean_calculator(versicolour),"   ","\n\n",'cov: ',"\n",cov_calculator(versicolour))
```

```
            MLE estimate:


    * setosa:

     mean:
     [4.26, 1.3259999999999998]

     cov:
     [[0.73574521 0.19258053]
     [0.19258053 0.15136837]]
    ______________________________

    * versicolour:

     mean:
     [5.552, 2.0259999999999994]

     cov:
     [[0.3132109  0.05169961]
     [0.05169961 0.07695137]]
```

```

#Visualize the 2-D joint pdf of petal length and width, a pdf function can be initialized by providing mean and covariance matrix
from scipy.stats import multivariate_normal
from matplotlib import cm
from matplotlib.ticker import LinearLocator


pdf_class0 = multivariate_normal(mean=mean_calculator(setosa), cov= cov_calculator(setosa))  #class0 = setosa
#pdf_class1 = multivariate_normal(mean=mean(versicolour), cov=cov(versicolour))  #class1 = versicolour
#Create a grid of x and y values on which to sample the pdf, this is done by providing a list of x-y of coordinates to the function pdf_class0.pdf(...)
#A 3D view of the pdf can be obtained using the function ax.plot_surface

#Code Example:

X = setosa_petal_length
Y = setosa_petal_width

X, Y = np.meshgrid(X, Y)
X_flat = X.flatten()
Y_flat = Y.flatten()
XY_list = np.concatenate((X_flat[:,np.newaxis],Y_flat[:,np.newaxis]),axis=1)
PDF_values = np.reshape(pdf_class0.pdf(XY_list), np.shape(X))

fig1, ax1 = plt.subplots(subplot_kw={"projection": "3d"},figsize=(4, 4), dpi=150)
ax1.plot_surface(X, Y, PDF_values, cmap=cm.coolwarm, alpha=0.7, linewidth=1)
plt.title('setosa 2-D joint pdf of petal length and width', fontsize=7)
plt.show


pdf_class1 = multivariate_normal(mean=mean_calculator(versicolour), cov = cov_calculator(versicolour))

XX = versicolour_petal_length
YY = versicolour_petal_width

XX, YY = np.meshgrid(XX, YY)
XX_flat = XX.flatten()
YY_flat = YY.flatten()
```

```
XYXY_list = np.concatenate((XX_flat[:,np.newaxis],YY_flat[:,np.newaxis]),axis=1)
PDF_values = np.reshape(pdf_class1.pdf(XYXY_list), np.shape(XX))

fig2, ax2 = plt.subplots(subplot_kw={"projection": "3d"},figsize=(4, 4), dpi=150)
ax2.plot_surface(XX, YY, PDF_values, cmap=cm.coolwarm, alpha=0.7, linewidth=1)
plt.title('versicolor 2-D joint pdf of petal length and width', fontsize=7)
plt.show
#To change the orientation of the 3D plot the function ax.view_init(), for a view from above select ax.view_init( 90, 0)

#After visualizing the pdf you can plot the points of the dataset on the estimated pdf using ax.scatter3D()
#For a better visualization of the points we suggest to make the pdf plot semi-transparent using the alpha parameter

#Code Example:
#PDF_points_class0 = pdf_class0.pdf(setosa_petal_length)
#plt.axes.scatter3D(setosa_petal_length, setosa_petal_width, PDF_points_class0, s=10)

#Note: the sample code was written only for class 0 but two plots have to be done, one for class 0 and one for class 1
#Note 2: the content of variables like Features_class0, Points_Class0_Feature0, Points_Class0_Feature1 should be substituted with variables created by
```

```
<function matplotlib.pyplot.show(*args, **kw)>
```



Student's comments to Exercise 1

1. The joint probability density functions of the two classes are depicted
2. Setosa shape has a problem in coding since the probability is more than 1!!!!
3. The figure is a 3D bellshape plane since it is the combination of 2 different bell-shape gaussian MLE
4. We can clearly see that the probability is maximum for the mean value of the data in both of the datasets

Exercise 2 - Model fitting for discrete distributions: Bag of Words

In this exercise, you will employ a real dataset (file *SMSSpamCollection*). The SMS Spam Collection v.1 (https://www.kaggle.com/datasets/uciml/sms-spam-collection-dataset) is a set of SMS tagged messages that have been collected for SMS Spam research. It contains one set of SMS messages in English of 5,574 messages, tagged acording being ham (legitimate) or spam. Task: you have to fit the parameters employed by a Naïve Bayes Classifier, using a Bernoulli model. Under this model, the parameters are:

- π_c, the prior probabilities of each class.
- θ_{jc}, the probability that feature j is equal to 1 in class c.

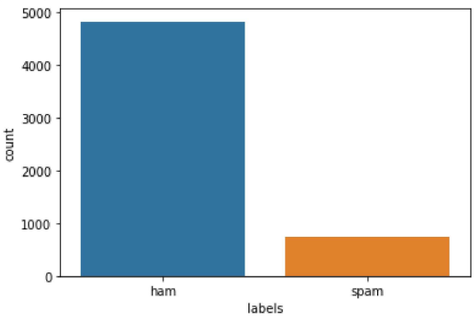
Model fitting can be done using the pseudocode at the end of the Lecture 3 slides.

Display the class-conditional densities θ_{jc1} and θ_{jc2}. Try to identify “uninformative” features (i.e., features j such that $\theta_{jc1} \simeq \theta_{jc2}$).

```
import numpy as np
import sklearn
from sklearn.feature_extraction.text import CountVectorizer
from sklearn import metrics
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns

#from NLP_base_library import text_splitting, print_dtm
import sys
```

```
import matplotlib.pyplot as plt

data=pd.read_csv('/content/SMSSpamCollection', encoding = "ISO-8859-1",sep='\t',header=None)
data.rename(columns = {0:'labels', 1:'text'}, inplace=True)

sns.countplot(data['labels'])

#Dataset Preprocessing: convesion from a collection of sentences into a Bag_of_words representation
from nltk.corpus import stopwords
from nltk.stem.porter import PorterStemmer
import re
import nltk

ps = PorterStemmer()
nltk.download('stopwords')

from nltk.corpus import stopwords
from nltk.stem.porter import PorterStemmer
ps = PorterStemmer()

corpus = []
for i in range(0, len(data)):
    review = re.sub('[^a-zA-Z]', ' ', data['text'][i]) #remove all characters which are not letters
    review = review.lower() #set everything to lower case
    review = review.split() #split into words
    #remove stopwords (articles and other uniformative words), replace words with their "stem" (basic form without conjugations and pluralizations )
    review = [ps.stem(word) for word in review if not word in stopwords.words('english')]
    review = ' '.join(review)
    corpus.append(review)

from sklearn.feature_extraction.text import CountVectorizer
cv = CountVectorizer(max_features=2500)



X = cv.fit_transform(corpus).toarray() #Convert dataset into Bag of words using the most informative 2500 words
feature_names = cv.get_feature_names_out()
print( f"Some examples of words which are used as features: {feature_names}") #This vector of strings contain the kind of words that are used as feature

y = pd.get_dummies(data['labels'])
y = y.iloc[:,1].values #convert the name of the two classes ("ham" and "spam") into numeric values "0" and "1"

#Partition Training and Testing set
#The training set is composed of 2000 SMS with 2500 features each, which identify wether a word from dictionary is present or not following the BoW repr
X_train = X[:2000,:]

#y_train is composed of the 2000 labels for the training examples, indicating wether a message is real (class 0) or spam (class 1)
y_train = y[:2000]

X_test = X[2000:3000,:] #1000 SMS are used in the testing set
y_test =y[2000:3000]
```

```
/usr/local/lib/python3.8/dist-packages/seaborn/_decorators.py:36: FutureWarning: Pass the following
  warnings.warn(
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
Some examples of words which are used as features: ['aah' 'aathi' 'abi' ... 'zed' 'zindgi' 'zoe']
(2000, 2500)
```




```
from google.colab import drive
drive.mount('/content/drive')
```

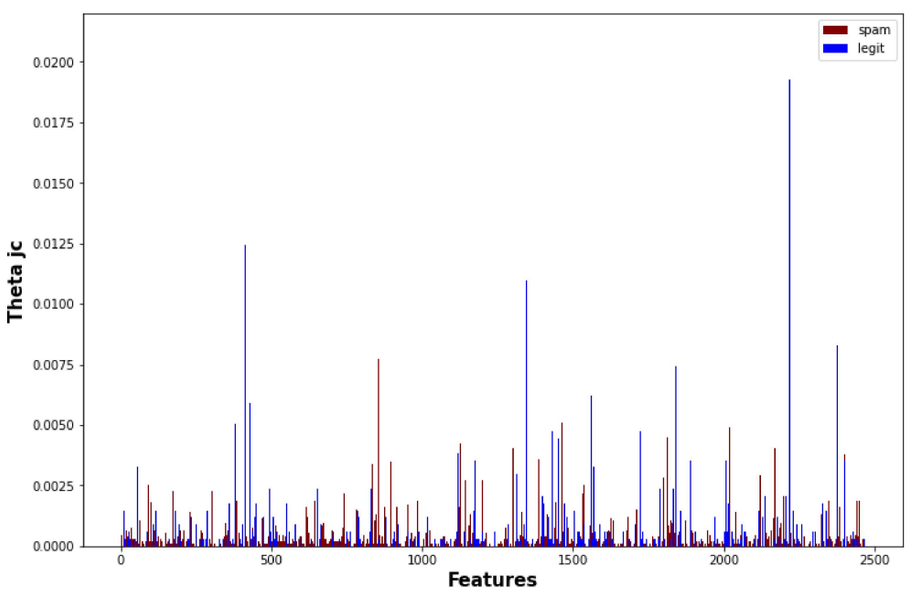
```
Mounted at /content/drive
```

```
#Evaluate the probabilities of the two classes, and the class conditional densities.
Nc_spam = 0  #Number of the spam messages (NC for spam)
Nc_ham = 0 #Number of the legit messages (NC for legit)
for i in range (len(y_train)):
    if y_train[i] == 1 : Nc_spam += 1
    else: Nc_ham += 1

ntotal = len(y_train) #Total number of the messages
prior_spam = Nc_spam/ntotal #Calculationg prior spam
prior_ham = Nc_ham/ntotal #Calculationg prior ham

# X (bag of word) contains the total number of the words (features)
# theta jc for the calss "spam" :


#claculating Njc of each class (Njc is a number for each feature and a vector of these numbers for a class)
Njc_class0 = []
Njc_class1 = []

for i in range(X_train.shape[1]): #Looping through columns(words/features) of X (word's BoW)
    Njc0 = 0
    Njc1 = 0
    for j in range (len(X_train[:,i])): #Looping through the elements of column i (word i) of matrix X
        if  X_train[:,i][j] == 1 and y_train[j] == 0 :
            Njc0 += 1
        if  X_train[:,i][j] == 1 and y_train[j] == 1 :
```

```
            Njc1 += 1
    Njc_class0.append(Njc0)
    Njc_class1.append(Njc1)

#calculating Nc of each class (Nc is a int number) (summing all the NJc for each class)
Nc_class0 = 0
Nc_class1 = 0

for i in range (len(Njc_class0)):
    Nc_class0 += Njc_class0[i]
for i in range (len(Njc_class1)):
    Nc_class1 += Njc_class1[i]

######print (Nc_class0 , Nc_class1)

######print (len(Njc_class0), Njc_class0)
######print (len(Njc_class1), Njc_class1)

#Calculating tetajc matrix by dividing each elemen of Njc by Nc
tetaj_class0 = np.divide(Njc_class0, Nc_class0)
tetaj_class1 = np.divide(Njc_class1, Nc_class1)

#######print (tetaj_class0)
# print (tetaj_class1)
# Finding uninformative tetajc by setting a minimum distance condition between tetaj_class 0 and tetaj_class1

uninformative = []
a = len(tetaj_class0)
i = 0
while i < a:
    if np.abs(tetaj_class0[i] - tetaj_class1[i]) <  (tetaj_class1[i]/10):
        uninformative.append(i)
        a -= 1
    i += 1


print('Total number of the uninformative features:', len(uninformative),
      "\nThe uninformative features' indexes:", uninformative,
      '\nTotal Number of informative features:', X_train.shape[1] - len(uninformative))


print('prior_spam:', prior_spam ,'prior_legit:', prior_ham)  # printing the prior of each class
```

```
Total number of the uninformative features: 32
The uninformative features' indexes: [165, 167, 194, 228, 264, 604, 607, 657, 703, 707, 798, 816, 840, 901, 937, 971, 983, 1262, 1299, 1362, 1456, 1476
Total Number of informative features: 2468
prior_spam: 0.14 prior_legit: 0.86
```

```
#Display the class-conditional densities θjc1 and θjc2.
barWidth = 1
fig = plt.subplots(figsize =(12, 8))
bin1 = np.arange(len(tetaj_class0))
bin2 = [x + barWidth for x in bin1]
plt.bar(bin1, tetaj_class0, color ='maroon', width = barWidth , label='spam')
plt.bar(bin2, tetaj_class1, color ='blue', width = barWidth, label = 'legit')
plt.xlabel('Features', fontweight ='bold', fontsize = 15)
plt.ylabel('Theta jc', fontweight ='bold', fontsize = 15)
plt.legend()
plt.ylim(0, 0.022)
plt.show()
```



Student's comments to Exercise 2

- Naive bayes classiffiers are generative and supervised classifiers which are based on bayes theorem. Since NBC assumes that features in a particular class are totally independent ▶ ***In text regognition problems, they can not understand the order of the words ***
- In BoW problems we consider the total space of the n words we have as the features (each word is a feature) and the documents to be classified (here SMS) is a n dimensional vector
- Here we have defined 2 classes (legit and spam)
- To model the data using NBC, We use bernouli distribution for fitting into the data to find prior and theta jc
- The figure shows $\theta jc0$ and $\theta jc1$ Which corresponds to the probability of the words belonging to each of the classes , It is clear that most of the word have a much higher chance to appear in a legit sms
- To find **uninformative features** we can set aa minimum distance condition between tetaj_class 0 and tetaj_class1 : $|\theta_{jc0} - \theta_j c_1| < \theta jc0$

Exercise 3 - Classification – discrete data

In this exercise, you will design a Naïve Bayes Classifier (NBC) for the Bag of Words (BoW) features for document classification that have been prepared in *Exercise 2*. In particular, in *exercise 2*, you have already estimated the following parameters:

- The prior probabilities of each class, $\pi_c = p(y = c)$.
- The class-conditional probabilities of each feature, $\theta_{jc} = p(x_j = 1|y = c)$.

These parameters have been estimated from the training data. In this exercise, you will use the test data, and classify each test vector using an NBC whose model has been fitted in *Exercise 2*. In particular, you will do the following:

- For each test vector, calculate the MAP estimate of the class the test vector belongs to. Remember: the MAP classifier chooses the class that maximizes $\max_c \log p(y = c|x) \propto \log p(x|y = c) + \log p(c)$. In the NBC, the features (i.e. each entry of x) are assumed to be statistically independent, so $p(x|y = c) = \prod_{j=1}^{D} p(x_j|y = c)$. This formula allows you to calculate $p(x|y = c)$ for a given test vector x using the parameters θ_{jc} already calculated in *Exercise 2*. Note that, after the logarithm, the product becomes a summation. It is much better to use the logarithm in order to avoid underflow.
- See how the accuracy changes when the prior is not taken into account (e.g. by comparing the MLE and MAP estimate).
- After classifying a test vector using the NBC, the obtained class can be compared with the truth (vector *ytest*).
- The accuracy of the classifier can be computed as the percentage of times that the NBC provides the correct class for a test vector.
- Repeat the same operations using the training data as test data, and compare the accuracy of the classifier on the training and test data.
- Note: It is expected that students implement the Naive Bayes classifier from scratch without using pre-made functions such as sklearn.naive_bayes

Optional:

If you plot the class-conditional densities as done at the end of Exercise 2, you will see that many features are uninformative; e.g., words that appear very often (or very rarely) in documents belonging to either class are not very helpful to classify a document. The NBC can perform a lot better if these uninformative features are disregarded during the classification, i.e. only a subset of the features, chosen among the most informative ones, are retained. To rank the features by "significance", one can employ the mutual information between feature x_j and class y (see Sec. 3.5.4 of the textbook):

$$I(X, Y) = \sum_{xj} \sum_{y} p(x_j, y) \log \frac{p(x_j, y)}{p(x_j)p(y)}$$

For binary features, the mutual information of feature j can be written as:

$$I_j = \sum_{c} \left[\theta_{jc}\pi_c \log \frac{\theta_{jc}}{\theta_j} + (1 - \theta_{jc})\pi_c \log \frac{1 - \theta_{jc}}{1 - \theta_j} \right]$$

with $\theta_j = p(x_j = 1) = \sum_c \pi_c \theta_{jc}$. For this part, you should:

- Calculate I_j for all features. Note: try to avoid divisions by zero adding the machine precision constant *eps* to the denominators.
- Rank the features by decreasing values of I_j, and keep only the K most important ones.
- Run the classifier employing only the K most important features, and calculate the accuracy.
- Plot the accuracy as a function of K.

```
#@title
# Evaluate the MAP on the test set
# MAP : Max log P(y=c|x) ~ log p(x|y =c) + log P(c)
# Calculating P(x|y=c) = n P(Xj|y=c)
#ham = 0     #spam = 1

MAP_sentences_0 = []
MAP_sentences_1 = []
MAP_sentence_i_0 = 0
MAP_sentence_i_1 = 0
############################ Test ##########################
for i in range(len(X_test)):                            # X_test : all of the test sentences (1000)
        pxj0 = 1
        pxj1 = 1
        for j in range (len(X_test[i])):                # Looping through the words of sentence i
            if X_test[i][j] == 1:                       # Checking if the word is available in the sentence
                if tetaj_class0[j] != 0 :
                    pxj0 *= tetaj_class0[j]                             # Calculating P(x|y=c) = n (Xj|y=c) of the class 0 sentence
                if tetaj_class1[j] != 0:
                    pxj1 *= tetaj_class1[j]                             # Calculating P(x|y=c) = n (Xj|y=c) of the class 1 sentence
        MAP_sentence_i_0 = np.log(pxj0) + np.log(prior_ham)         # Calculating MAP of the sentence i for class 0 (ham) : Max log P(y=c|x) ~ log p(x|y
        MAP_sentences_0.append(MAP_sentence_i_0)                     # Creating a vector of Class 0 Map estimate of each sentence
        MAP_sentence_i_1 = np.log(pxj1) + np.log(prior_spam)        # Calculating MAP of the sentence i for class 1 (spam) : Max log P(y=c|x) ~ log p(x
        MAP_sentences_1.append(MAP_sentence_i_1)                     # Creating a vector of Class 1 Map estimate of each sentence
y_hat_test =[]
#print ('MAP0',MAP_sentences_0)
#print ('MAP1',MAP_sentences_1)

# Calculating MAP estimate of each test sentence (y_hat_test):

for i in range (len(MAP_sentences_0)):
    if np.abs(MAP_sentences_0 [i]) >  np.abs(MAP_sentences_1 [i]):
        y_hat_test.append(0)
    if np.abs(MAP_sentences_0 [i]) <  np.abs(MAP_sentences_1 [i]):
        y_hat_test.append(1)

# calculating the accuracy of estimation by comparing y_hat_test and y_test :
neq=0
for i in range (len(y_test)):
    if y_test[i] == y_hat_test[i]:
        neq +=1

print('The Accuracy for MAP on test is equal to:' ,neq/len(y_test)*100 ,'%')


############################ Train ##########################

for i in range(len(X_train)):                             # X_train : all of the traint sentences (2000)
        pxj0 = 1
        pxj1 = 1
```

```
        for j in range (len(X_train[i])):                # Looping through the words of sentence i
            if X_train[i][j] == 1:                       # Checking if the word is available in the sentence
                if tetaj_class0[j] != 0 :
                    pxj0 *= tetaj_class0[j]                          # Calculating P(x|y=c) = n (Xj|y=c) of the class 0 sentence
                if tetaj_class1[j] != 0:
                    pxj1 *= tetaj_class1[j]                          # Calculating P(x|y=c) = n (Xj|y=c) of the class 1 sentence
        MAP_sentence_i_0 = np.log(pxj0) + np.log(prior_ham)         # Calculating MAP of the sentence i for class 0 (ham) : Max log P(y=c|x) ~ log p(x|y
        MAP_sentences_0.append(MAP_sentence_i_0)                     # Creating a vector of Class 0 Map estimate of each sentence
        MAP_sentence_i_1 = np.log(pxj1) + np.log(prior_spam)        # Calculating MAP of the sentence i for class 1 (spam) : Max log P(y=c|x) ~ log p(x
        MAP_sentences_1.append(MAP_sentence_i_1)                     # Creating a vector of Class 1 Map estimate of each sentence
y_hat_train =[]
#print ('MAP0',MAP_sentences_0)
#print ('MAP1',MAP_sentences_1)

# Calculating MAP estimate of each test sentence (y_hat_train):

for i in range (len(MAP_sentences_0)):
    if np.abs(MAP_sentences_0 [i]) >  np.abs(MAP_sentences_1 [i]):
        y_hat_train.append(0)
    if np.abs(MAP_sentences_0 [i]) <  np.abs(MAP_sentences_1 [i]):
        y_hat_train.append(1)

# calculating the accuracy of estimation by comparing y_hat_test and y_test :
neq=0
for i in range (len(y_train)):
    if y_train[i] == y_hat_train[i]:
        neq +=1

print('The Accuracy for MAP on train is equal to:' ,neq/len(y_train)*100 ,'%')
```

```
    The Accuracy for MAP on test is equal to: 82.6 %
    The Accuracy for MAP on train is equal to: 70.25 %
```

```
#Evaluate the MLE on the test set

MLE_sentences_0 = []
MLE_sentences_1 = []
MLE_sentence_i_0 = 0
MLE_sentence_i_1 = 0
############################ Train #######################################
for i in range(len(X_test)):                             # X_test : all of the test sentences (1000)
        pxj0 = 1
        pxj1 = 1
        for j in range (len(X_test[i])):                 # Looping through the words of sentence i
            if X_test[i][j] == 1:                        # Checking if the word is available in the sentence
                if tetaj_class0[j] != 0 :
                    pxj0 *= tetaj_class0[j]                          # Calculating P(x|y=c) = n (Xj|y=c) of the class 0 sentence
                if tetaj_class1[j] != 0:
                    pxj1 *= tetaj_class1[j]                          # Calculating P(x|y=c) = n (Xj|y=c) of the class 1 sentence
        MLE_sentence_i_0 = np.log(pxj0)        # Calculating MLE of the sentence i for class 0 (ham) : Max log P(y=c|x) ~ log p(x|y =c)  ||| **Without p
        MLE_sentences_0.append(MLE_sentence_i_0)                     # Creating a vector of Class 0 Map estimate of each sentence
        MLE_sentence_i_1 = np.log(pxj1)        # Calculating MLE of the sentence i for class 1 (spam) : Max log P(y=c|x) ~ log p(x|y =c) ||| **Without p
        MLE_sentences_1.append(MLE_sentence_i_1)                     # Creating a vector of Class 1 Map estimate of each sentence
y_hat_test =[]
#print ('MAP0',MAP_sentences_0)
#print ('MAP1',MAP_sentences_1)

# Calculating MAP estimate of each test sentence (y_hat_test):

for i in range (len(MLE_sentences_0)):
    if np.abs(MLE_sentences_0 [i]) >  np.abs(MLE_sentences_1 [i]):
        y_hat_test.append(0)
    if np.abs(MLE_sentences_0 [i]) <  np.abs(MLE_sentences_1 [i]):
        y_hat_test.append(1)
    if np.abs(MLE_sentences_0 [i]) ==  np.abs(MLE_sentences_1 [i]): # The SMS with the same MLE for both classes are considered as legit
        y_hat_test.append(0)



# calculating the accuracy of estimation by comparing y_hat_test and y_test :
neq=0
for i in range (len(y_test)):
    if y_test[i] == y_hat_test[i]:
        neq +=1

print('The Accuracy for MLE Test is equal to:' ,neq/len(y_test)*100 ,'%')

############################ Test#######################################
for i in range(len(X_train)):                            # X_train : all of the test sentences (1000)
        pxj0 = 1
        pxj1 = 1
        for j in range (len(X_train[i])):                # Looping through the words of sentence i
            if X_train[i][j] == 1:                       # Checking if the word is available in the sentence
                if tetaj_class0[j] != 0 :
                    pxj0 *= tetaj_class0[j]                          # Calculating P(x|y=c) = n (Xj|y=c) of the class 0 sentence
                if tetaj_class1[j] != 0:
                    pxj1 *= tetaj_class1[j]                          # Calculating P(x|y=c) = n (Xj|y=c) of the class 1 sentence
        MLE_sentence_i_0 = np.log(pxj0)        # Calculating MLE of the sentence i for class 0 (ham) : Max log P(y=c|x) ~ log p(x|y =c)  ||| **Without p
        MLE_sentences_0.append(MLE_sentence_i_0)                     # Creating a vector of Class 0 Map estimate of each sentence
        MLE_sentence_i_1 = np.log(pxj1)        # Calculating MLE of the sentence i for class 1 (spam) : Max log P(y=c|x) ~ log p(x|y =c) ||| **Without p
        MLE_sentences_1.append(MLE_sentence_i_1)                     # Creating a vector of Class 1 Map estimate of each sentence
y_hat_train =[]
#print ('MAP0',MAP_sentences_0)
#print ('MAP1',MAP_sentences_1)

# Calculating MAP estimate of each train sentence (y_hat_train):

for i in range (len(MLE_sentences_0)):
    if np.abs(MLE_sentences_0 [i]) >  np.abs(MLE_sentences_1 [i]):
        y_hat_train.append(0)
    if np.abs(MLE_sentences_0 [i]) <  np.abs(MLE_sentences_1 [i]):
        y_hat_train.append(1)
    if np.abs(MLE_sentences_0 [i]) ==  np.abs(MLE_sentences_1 [i]): # The SMS with the same MLE for both classes are considered as legit
        y_hat_train.append(0)

```

```python


# calculating the accuracy of estimation by comparing y_hat_train and y_train :
neq=0
for i in range (len(y_train)):
    if y_train[i] == y_hat_train[i]:
        neq +=1

print('The Accuracy for MLE Train is equal to:' ,neq/len(y_train)*100 ,'%')
```

```
The Accuracy for MLE Test is equal to: 85.9 %
The Accuracy for MLE Train is equal to: 73.1 %
```

Student's comments to Exercise 3

Comparing accuracy of the Test and Train:

- The Accuracy of test dataset is more than the accuracy for the training data set, if the difference was small there was no problem, but in this situation with more than 10% difference, it seems that we have underfitting. Maybe happend due to not using all of the feature that is in the data!!!!

Comparing the accuracy of MLE and MAP:

- The Accuracy of MLE is a bit higher than MAP for both the test and training dataset ! We expected more accuracy for MAP since it also observes prior in addition to the likelihood but sometime in large dataset MLE can become becomes more robust and less sensitive to the choice of prior

---

Exercise 4 – Plotting the ROC curve

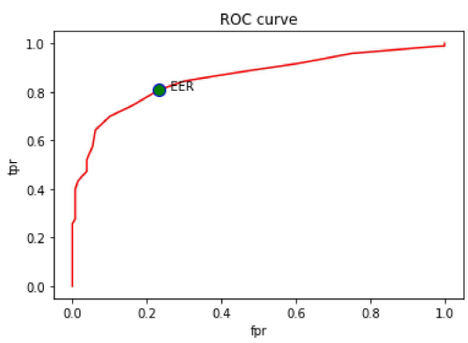
For the discrete data classification problem of Exercise 3, analyse the performance of the classifier plotting the complete ROC curve, instead of simply measuring the accuracy. This requires to do the following.

- Instead of classifying the documents choosing class 1 if $p(y=1|x) > p(y=2|x)$, now you can generalize this to choosing class 1 if $\frac{p(y=1|x)}{p(y=2|x)} > \tau \in [0, \infty)$ for some threshold τ that determines the compromise between true positive rate (TPR) and false positive rate (FPR).
- Choose a **reasonable** range of values for τ. For each value of τ, compute the TPR and FPR on the dataset (Hint: determine suitable minimum and maximum values for τ, and sample densely enough in that range).
- Plot a curve of TPR as a function of FPR – this is the ROC curve for this classifier.
- Determine an estimate of the Equal Error Rate (EER), i.e. the point of the ROC curve such that TPR+FPR=1.

```python
from tkinter import N
#Use the posterior probabilities previously and classify using the formula above to estimate tau

# Calculating MAP estimate of each test sentence (y_hat_test):

def ROC_estimate (a,b,c):
    mapclass0 = a
    mapclass1 = b
    taw=c

    y_hat_test_ROC = []
    for i in range (len(MAP_sentences_0)):
        if np.abs(mapclass0 [i]) / np.abs(mapclass1[i]) > taw:
            y_hat_test_ROC.append(0)
        if np.abs(mapclass0 [i]) / np.abs(mapclass1[i]) < taw:
            y_hat_test_ROC.append(1)

    return y_hat_test_ROC

# calculating the accuracy of estimation by comparing y_hat_test and y_test :

def Accuracy (a,b):
    y = a
    yhat = b
    neq=0
    for i in range (len(y)):
        if y[i] == yhat[i]:
            neq +=1
    acc = neq/len(y_test)*100
    return acc




# Function to measure TPR
def TPR (a,b): #based on "spam as the true"
    test = a # test is what the classifier says about the class of the test points in the test dataset
    true = b # true is the correct classes of the test points in the test dataset
    ntrue = 0
    for i in range (len(true)): # Calculating the total number of y=1
        if true [i] == 0:
            ntrue += 1

    npos = 0
    for i in range (len(true)):
        if true [i] == 0 and  test [i] == 0:
            npos += 1

    tpr = npos / ntrue
    return tpr

""""""""""""""""""""""""""""""""""""""""""""""""""""""
# Function to measure FPR
def FPR (a,b): #based on "spam as the true"
    test = a  # test is what the classifier says about the class of the test points in the test dataset
    true = b  # true is the correct classes of the test points in the test dataset
    ntrue = 0
    for i in range (len(true)): #calculating the total number of y=0
```

```
        if true [i] == 1:
            ntrue += 1

    npos = 0
    for i in range (len(true)):
        if true [i] == 1 and  test [i] == 0:
            npos += 1

    fpr = npos / ntrue
    return fpr

tpr = TPR (ROC_estimate(MAP_sentences_0, MAP_sentences_1, 15) , y_test )

fpr = FPR (ROC_estimate(MAP_sentences_0, MAP_sentences_1, 15) , y_test )

# Generating a range of taw for calculating tpr and fpr based on the different amounts of taw
taw_range = np.arange(0,18,0.1)
# Calculating tpr and fpr for different values of taw
tprs = []
fprs = []
EER = []
EERpoint = []

for i in taw_range:
    tpr = TPR (ROC_estimate(MAP_sentences_0, MAP_sentences_1, i) , y_test )
    tprs.append(tpr)
    fpr = FPR (ROC_estimate(MAP_sentences_0, MAP_sentences_1, i) , y_test )
    fprs.append(fpr)
    # Finding EER point where tpr + fpr = 1
    if np.abs(TPR (ROC_estimate(MAP_sentences_0, MAP_sentences_1, i) , y_test ) + FPR (ROC_estimate(MAP_sentences_0, MAP_sentences_1, i) , y_test) - 1) < 0.05:
        EER.append(i) #saving the taw of EER
        EERpoint.append (FPR (ROC_estimate(MAP_sentences_0, MAP_sentences_1, i) , y_test))   # saving the FPR in which EER happens
        EERpoint.append (TPR (ROC_estimate(MAP_sentences_0, MAP_sentences_1, i) , y_test ))  # saving the TPR in which EER happens


# printing the data
print('\n')
print ('The taw in which EE happens: ', EER[0])
print('\n')
print('FPR and TPR of taw = EER is equal to :',EERpoint)
print('\n')
print('The Accuracy for taw = EER is equal to:' ,Accuracy(y_test , ROC_estimate(MAP_sentences_0, MAP_sentences_1, EER[0])) ,'%')

# plotting the ROC curve
print('\n\n')
plt.plot(fprs,tprs, color = "r")
plt.xlabel("fpr")
plt.ylabel("tpr")
plt.title('ROC curve')
plt.plot(EERpoint[0], EERpoint[1], marker="o", markersize=10, markeredgecolor="blue", markerfacecolor="green")
plt.annotate("   EER", (EERpoint[0], EERpoint[1]))
plt.show()


#Note: To estimate the TPR you need to compute the number of cases where class 1 is correctly predicted,
#this value then has to be divided by the number of elements in the test set which belong to class 1

#The FPR is computed by selecting the number of cases where class 1 is predicted incorrectly,
#this value then has to be divided by the number of elements in the test set which do not belong to class 1

#The point corresponding to the EER can be found by plotting on the ROC curve the function y = 1 - x
```

```
The taw in which EE happens:  1.1


FPR and TPR of taw = EER is equal to : [0.23255813953488372, 0.8082663605051664]


The Accuracy for taw = EER is equal to: 80.30000000000001 %
```



Student's comments to Exercise 4

- Most of the comments related to the code are added between the code lines
- EER point indicates the threshold value where the classifier makes the same number of false positive and false negative predictions
- curve is almost acceptabe and near to the top left of the figure (High AUC)
- The EER point is founded by putting an if condition $|TPR + FPR - 1| < 0.05$ (small range) in the loop of creating the ROC values

Exercise 5 – Classification – continuous data

This exercise employs the Iris dataset already employed in Exercise 1, and performs model fitting and classification using several versions of **Gaussian discriminative analysis**. However, for this exercise the available data have to be divided into two sets, namely *training* and *test* data.

You will have to 1) re-fit the training data to the specific model (see below), 2) classify each of the test samples, and 3) calculate the accuracy of each classifier.

Classifiers to be employed:

- Two-class quadratic discriminant analysis (fitting: both mean values and covariance matrices are class-specific – same as in exercise 1).
- Two-class linear discriminant analysis (fitting: class-specific mean values as in the previous case. Shared covariance matrix is calculated putting together the elements of both classes; the mean values should also be recalculated accordingly).

```
import numpy as np
import matplotlib.pyplot as plt
import h5py

Dataset1 = h5py.File('/content/Lab2_Ex_1_Iris.hdf5')
Data = np.array(Dataset1.get('Dataset'))

Train = Data[:50,:]
Test = Data[50:,:]

"""************* Creating a training and a test for each of the classes  **************"""

#Separate the training dataset into the two classes
#Create filters
indx0 = np.where(Train[:,2] == 0)                                    # Returns a list of indices of the rows which their third element is equal to zero(cr
indx1 = np.where(Train[:,2] == 1)                                    # """""""""""""""""""""""""""""" is equal to one
# Apply filters
setosa_train = Train[indx0][:,0:2]                                   # uses the index0 list to separate (filter) the rows which their third element is equ
versicolour_train = Train[indx1][:,0:2]                              # """""""""""""""""""""""""""""""

#Separate the test dataset into the two classes
#Create filters
indx0 = np.where(Test[:,2] == 0)                                    # Returns a list of indices of the rows which their third element is equal to zero(cre
indx1 = np.where(Test[:,2] == 1)                                    # """""""""""""""""""""""""""""" is equal to one
# Apply filters
setosa_test = Test[indx0][:,0:2]                                    # uses the index0 list to separate (filter) the rows which their third element is equa
versicolour_test = Test[indx1][:,0:2]                               # """""""""""""""""""""""""""""""

"""************* Defining Functions to calculate mean and cov  ***************"""

def mean (x):
    x_mean = [0,0]
    for j in range(x.shape[1]):
        for i in range(x.shape[0]):
            x_mean[j] += x[i,j]
        x_mean[j] /= x.shape[0]
    return x_mean

def cov(y):
    y_mean= mean(y)
    y_mean = np.array(y_mean)
    y_cov = np.empty([len(y.shape), len(y.shape)])
    for i in range(y.shape[0]):
        y_cov +=  (np.subtract( y[i][:, np.newaxis] @ y[i][:, np.newaxis].T , y_mean[:, np.newaxis] @ y_mean[:, np.newaxis].T))
    return np.divide(y_cov,y.shape[0])

"""*************  Calculating mean and cov  ***************"""

# class specified means
mean_setosa_train = mean (setosa_train)
mean_versicolour_train = mean (versicolour_train)
   #print ('mean_setosa_train:', mean_setosa_train)
   #print ('mean_versicolour_train:', mean_versicolour_train)
# class specified covariance
cov_setosa_train = cov (setosa_train)
cov_versicolour_train = cov (versicolour_train)
   #print ('cov_setosa_train:\n', cov_setosa_train)
   #print ('cov_versicolour_train:\n', cov_versicolour_train)
# shared covariance
train_classless = Train[:,0:2]
cov_train = cov(train_classless)
   #print ('cov_train:\n', cov_train)

"""************* 1.Quadratic Discriminative Analysis ***************"""

# Since the priors for setosa & versicolour is uniform we can use "Nearest Centroid" classifiers

# Defining nearest centroid classifier for QDA
def nearest_centroid (a):
    D = 1
    x = a

    x_minus_mean = np.subtract(x,mean_setosa_train)
    x_minus_mean_transpose = x_minus_mean.T
    cov_inverse_setosa = np.linalg.inv(cov_setosa_train)
    D_setosa = x_minus_mean_transpose @ cov_inverse_setosa @ x_minus_mean

    x_minus_mean = np.subtract(x,mean_versicolour_train)
    x_minus_mean_transpose = x_minus_mean.T
    cov_inverse_versicolour = np.linalg.inv(cov_versicolour_train)
    D_versicolour = x_minus_mean_transpose @ cov_inverse_versicolour @ x_minus_mean

    if D_setosa > D_versicolour:
        a=1
    else:
        a=0
    return a

# Calculating the estimation vector
Test1 = Test[:,0:2]
estimated_calss_quadratic = []
for i in range(len(Test)):
    estimated_calss_quadratic.append(nearest_centroid(Test1[i]))

# Calculating the accuracy
```

```
num = 0
for i in range (len(Test)):
    if Test[i,2] == estimated_calss_quadratic[i]:
        num +=1

print ("Accuracy of Quadratic Discriminative Analysis : ", num/len(Test))


""" ************* 2.Linear Discriminative Analysis ***************"""

# We use discriminant function

def discriminant (a):
    x = a
    meansetosa_minus_meanversicolour = np.subtract(mean_setosa_train, mean_versicolour_train)
    cov_total_inverse = np.linalg.inv(cov_train)
    discriminant_x = meansetosa_minus_meanversicolour.T @ cov_total_inverse @ x

    return discriminant_x


meansetosa_plus_meanversicolour = np.add(mean_setosa_train, mean_versicolour_train)


def linear_discriminant_classifier(a):
    x = a
    clas = 2
    if discriminant(x) < 0.5* discriminant(meansetosa_plus_meanversicolour):
        clas = 1
    if discriminant(x) > 0.5* discriminant(meansetosa_plus_meanversicolour):
        clas = 0

    return clas


# Calculating the estimation vector
Test1 = Test[:,0:2]
estimated_calss_linear = []
for i in range(len(Test)):
    estimated_calss_linear.append(linear_discriminant_classifier(Test1[i]))

# Calculating the accuracy
num = 0
for i in range (len(Test)):
    if Test[i,2] == estimated_calss_linear[i]:
        num +=1

print ("Accuracy of Linear Discriminative Analysis : ", num/len(Test))



```

```
Accuracy of Quadratic Discriminative Analysis :  0.92
Accuracy of Linear Discriminative Analysis :  0.9
```

Student's comments to Exercise 5

Quadratic Analysis

The classification function in QDA is:

$$D_i^2(y) = (y - \bar{y}_i)' S_i^{-1} (y - \bar{y}_i), \qquad i = 1, 2, \cdots, k$$

The observation y is assigned to the group for which $D2i(y)$ is smallest. **Note:** In QDA the covariance matrix is class specified

We define the function based on the formula and calculate the Accuracy

Linear Anayisus

In LDA, The classification procedure assigns the observation vector y to group 1 if its discriminant function $z = a'y$ is closer to $z1$ or group 2 if its discriminant function is closer to z

$$z = a'y = (\bar{y}_1 - \bar{y}_2)' S_{p1}^{-1} y$$

$$z > \frac{1}{2}(\bar{z}_1 + \bar{z}_2)$$

We can now express the classification function regarding the observation vector y . $\frac{1}{2}(\bar{z}_1 + \bar{z}_2) = \frac{1}{2}(\bar{y}_1 - \bar{y}_2)' S_{p1}^{-1} (\bar{y}_1 + \bar{y}_2)$

Thus the observation vector y is assigned to a group determined by the following:

Assign y to group 1 if:

$$a'y = (\bar{y}_1 - \bar{y}_2)' S_{p1} y > \frac{1}{2}(\bar{y}_1 - \bar{y}_2)' S_{p1}^{-1} (\bar{y}_1 + \bar{y}_2)$$

Or assign y to group 2 if:

$$a'y = (\bar{y}_1 - \bar{y}_2)' S_{p1} y < \frac{1}{2}(\bar{y}_1 - \bar{y}_2)' S_{p1}^{-1} (\bar{y}_1 + \bar{y}_2)$$

$S_{p1}y$ is the shared covariance matrix

Conclusion: Although QDA performed 2% better due to its flexibility but it might also overfit in some cases

---

Exercise 6 – Classification – continuous data

Classify the data in the phoneme dataset from Lab. 1 using quadratic discriminant analysis, linear discriminat analysis and a Naive Bayes classifier.

Compute the accuracy of each classifier and compare its performance with that of the k-nn classifier developed in Lab. 1.

Note: in Lab1 we had to employ a subset of the oiginal dataset due to the fact that k-nn has a quadratic complexity making it unfit for use on large datasets. The algorithms illustrated in this Lab have smaller complexity and thus it is possible to train on more data.

For this exercise you can use the sklearn library.

```
import numpy as np
import h5py
from scipy.stats import multivariate_normal
```

```
Dataset2 = h5py.File("/content/Lab2_Ex_6_phoneme.hdf5")
Data = np.array(Dataset2.get('Dataset'))

Train = Data[:4000,:]
Test = Data[4000:,:]
len_dat = np.shape(Test)[0]
Train_x = Train[:,:255]
Train_y = Train[:,256]
Test_x = Test[:,:255]
Test_y = Test[:,256]

from sklearn import metrics
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis, QuadraticDiscriminantAnalysis
from sklearn.naive_bayes import GaussianNB

#Part 1
clf = LinearDiscriminantAnalysis()
clf.fit(Train_x, Train_y)
linear_predict = clf.predict(Test_x)

#Complete here

#Part 2
clf2 = QuadraticDiscriminantAnalysis()
clf2.fit(Train_x, Train_y)
Quadratic_predict = clf2.predict(Test_x)
#Complete here

#Part 3
clf3 = GaussianNB()
clf3.fit(Train_x, Train_y)
Gaussian_predict = clf3.predict(Test_x)
#Complete here


# Accuracy Function
def Acc (a,b):
    predicted = a
    correct = b
    num = 0
    for i in range(len(predicted)):
        if predicted[i] == correct[i] :
            num += 1

    acc = num / len(predicted)
    return acc

# Calculating Accuracy
# LinearDiscriminantAnalysis
Linear_acc = Acc (linear_predict , Test_y )
print ("The accuracy of LinearDiscriminantAnalysis on the test data is equal to : (%) \n " , Linear_acc*100)

# QuadraticDiscriminantAnalysis
Quadratic_acc = Acc (Quadratic_predict , Test_y )
print ("The accuracy of QuadraticDiscriminantAnalysis on the test data is equal to : (%) \n " , Quadratic_acc*100)

# GaussianDiscriminantAnalysis
Gaussian_acc = Acc (Gaussian_predict , Test_y )
print ("The accuracy of GaussianDiscriminantAnalysis on the test data is equal to : (%) \n " ,Gaussian_acc*100)



```

```
The accuracy of LinearDiscriminantAnalysis on the test data is equal to : (%)
  94.10609037328095
The accuracy of QuadraticDiscriminantAnalysis on the test data is equal to : (%)
  88.99803536345776
The accuracy of GaussianDiscriminantAnalysis on the test data is equal to : (%)
  89.9803536345776
```

Student's comments to Exercise 6

Compared to KNN classifier in lab 1 with 96% Accuracy of the test data there's a lower accuracy !!! KNN was far better!

The LDA performed better and the probable reason is that the Gaussian assumptions holds, meaning that the class distributions are well-separated and approximately Gaussian, hence LDA can provide better accuracy than QDA.

STATISTICAL LEARNING AND NEURAL NETWORKS, A.A. 2022/2023

COMPUTER LAB 3 - Principal component analysis

Duration: 3 hours

Introduction:

Hyperspectral images are scientific images of the Earth, acquired by satellites or aicrafts; rather than having three R/G/B color channels, these images have a lot more "color" components obtained through a fine sampling of the wavelength (hence the name "hyper"-spectral). The resulting 3-dimensional dataset has one image (spectral band or "color") for every sampled wavelength, which represents the measured radiance from each pixel at that specific wavelength. Hyperspectral images are very useful for image analysis. For every pixel at a given spatial position, it is possible to extract a so-called spectral vector, i.e. the 1-dimensional vector of values assumed by that pixel at all wavelengths. Assuming that each pixel is composed of just one substance, the spectral vector represents the radiance of that substance at all the wavelengths that have been sampled. Spectral vectors, therefore, can be used to infer which substance is contained in a given pixel – a typical classification problem that has a lot of practical applications in agriculture, analysis of land use / land cover, and other applications related to the study of the environment.

In this lab you will use a real hyperspectral image that has been acquired by the AVIRIS instrument, an airborne hyperspectral imager operated by the NASA. The image represents a scene of Indian Pines (Indiana, USA). It has a size of 145x145 pixels and 220 spectral bands. Along with the image, a ground truth is available, in terms of labels specifying which class (out of 16) each pixel belongs to. The classes are reported below; for more information, please see http://www.ehu.eus/ccwintco/index.php/Hyperspectral_Remote_Sensing_Scenes#Indian_Pines

#	Class	Samples
1	Alfalfa	46
2	Corn-notill	1428
3	Corn-mintill	830
4	Corn	237
5	Grass-pasture	483
6	Grass-trees	730
7	Grass-pasture-mowed	28
8	Hay-windrowed	478
9	Oats	20
10	Soybean-notill	972
11	Soybean-mintill	2455
12	Soybean-clean	593
13	Wheat	205
14	Woods	1265
15	Buildings-Grass-Trees-Drives	386
16	Stone-Steel-Towers	93

The purpose of this computer lab is twofold: To apply PCA to the spectral vectors in order to reduce their dimensionality.

- To apply PCA to the spectral vectors in order to reduce their dimensionality.
- To perform classification on the reduced data (optional)

Exercise 1 – PCA

In this exercise, you will employ the Indian Pines dataset. You will not do this for the entire dataset, but only for the spectral vectors belonging to **two classes** (as in the optional exercise you will perform 2-class classification on the PCA coefficients).

Reminder: the input to the PCA must always have zero mean: besides the sample covariance, you will have to compute the **mean value μ** over the training set and subtract it from each test vector before applying PCA.

Task: You have to reduce the dimensionality of the spectral vectors of the two classes you have chosen using PCA. In particular, you should perform the following:

- Extract spectral vectors of two classes, as described above (see sample code below).
- Estimate the sample covariance matrix of the dataset as a whole (i.e., considering together spectral vectors of the two classes)
- Perform the eigenvector decomposition of the sample covariance matrix. You can use the numpy linalg.eig function, which outputs the matrix containing the eigenvectors as columns, and a diagonal matric containing the eigenvalues on the main diagonal.
    - Note: in the output matrix, eigenvectors/eigenvalues are not necessarily ordered by eigenvalue magnitude. You should sort them by yourself.
- Choose a number of dimensions K<=220.
- Construct the eigenvector matrix W for K components (i.e., select the last K columns)
- Using W, compute the PCA coefficients for each spectral vector in the data set
- Then from the PCA coefficients obtain an approximation of the corresponding vector and compute the error (mean square error - MSE)
- Plot the average MSE over the test set as a function of K.
- Plot the eigenvectors corresponding to the 3 largest eigenvalues – this will give you an idea of the basis functions employed by PCA

```
import numpy as np
import h5py
import scipy.io
import pandas as pd

mat = scipy.io.loadmat('Indian_pines.mat')
indian_pines = np.array(mat['indian_pines'])
print(indian_pines.shape,len(indian_pines[0]))

mat = scipy.io.loadmat('Indian_pines_gt.mat')
indian_pines_gt = np.array(mat['indian_pines_gt'])
print (indian_pines_gt.shape ,indian_pines_gt)
```

```
(145, 145, 220) 145
(145, 145) [[3 3 3 ... 0 0 0]
 [3 3 3 ... 0 0 0]
 [3 3 3 ... 0 0 0]
 ...
```

```
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]]
```

Extract spectral vectors of two classes, as described above (see sample code below).

```
# Coosing Class 4 , Corn , 237 samples
class1_value = 4 # corn

class1=np.zeros((1500,220))                       # Create an empty 2d array to store spectral data of each of the classes
n=0
for i in range(145):                              # Checking all of the pixels
  for j in range(145):
    if indian_pines_gt[i,j]== class1_value:       # Checking if the pixel belongs to the specified class
      class1[n,:] = indian_pines[i,j,:]/1.        # Adding the pixels's spectral values row vector (1*220) to the empty class1 array
      n=n+1                                       # Counting the number of the pixels which belonged to the class (n)
class1=class1[:n,:]                               # Slicing the first not empty (n) rows of the class 1, resulting in a (n*220) array

# Choosing Class 14 , Woods , 1265 samples
class2_value = 14 # woods

class2=np.zeros((1500,220))                       # Create an empty 2d array to store spectral data of each of the classes
n=0
for i in range(145):                              # Checking all of the pixels
  for j in range(145):
    if indian_pines_gt[i,j]== class2_value:       # Checking if the pixel belongs to the specified class
      class2[n,:] = indian_pines[i,j,:]/1.        # Adding the pixels's spectral values row vector (1*220) to the empty class1 array
      n=n+1                                       # Counting the number of the pixels which belonged to the class (n)
class2=class2[:n,:]                               # Slicing the first not empty (n) rows of the class 1, resulting in a (n*220) array

print (class1.shape , class1)
print (class2.shape , class2)
```

```
(237, 220) [[3170. 4001. 4330. ... 1024. 1019. 1004.]
 [3167. 4005. 4423. ... 1038. 1015. 1000.]
 [3002. 4133. 4503. ... 1031. 1014. 1013.]
 ...
 [3705. 4497. 4785. ... 1035. 1013. 1024.]
 [3868. 4373. 4777. ... 1029. 1009. 1014.]
 [2592. 4369. 4599. ... 1038. 1013. 1019.]]
(1265, 220) [[2741. 4135. 4147. ...  998. 1001. 1010.]
 [2567. 4135. 4324. ... 1005. 1005. 1000.]
 [3167. 4128. 4320. ... 1001. 1000. 1004.]
 ...
 [2725. 3982. 4099. ... 1004. 1008. 1009.]
 [3320. 3862. 4015. ... 1004. 1000. 1009.]
 [3154. 3986. 4099. ...  996. 1003. 1000.]]
```

Estimate the sample covariance matrix of the dataset as a whole (i.e., considering together spectral vectors of the two classes)

```
# Insert code here
bothclass = np.concatenate((class1,class2) ,axis=0)        # Concatenating the data of the classes

# Converting to DataFrame
bothclass_df = pd.DataFrame (bothclass)                     # Convert to a dataframe to be able to get mean

# Shuffling the data
bothclass_df = bothclass_df.sample(frac = 1)


# Creating the test and training datasets
bothclass_test_df = bothclass_df.iloc[1000:1502 , :]       # Choose the last 502 points(pixels) as the test dataset
bothclass_test = bothclass_test_df.to_numpy()              # ~502*220
bothclass_df = bothclass_df.iloc[:1000 , :]                # Choose the first 1000 points as the training dataset
bothclass = bothclass_df.to_numpy()

# Normalizing the training data
mean_bothclass_df = bothclass_df.mean()                    # Calcualte the mean
mean_bothclass = mean_bothclass_df.to_numpy()              # Convert back to an array   ~(220,)
bothclass -= mean_bothclass                                # subtracting mean vector from all data (all rows)
bothclass = bothclass.T                                    # Reshaping the data ~220*1000

# Normalizing the test data
bothclass_test -= mean_bothclass
bothclass_test = bothclass_test.T                          # ~220*502        # Normalized


# Calculating cov matrix based on the training matrix (bothclass)
cov_bothclass = np.cov(bothclass)                          # ~220*220
```

Double-click (or enter) to edit

Perform the eigenvector decomposition of the sample covariance matrix. You can use the numpy linalg.eig function, which outputs the matrix containing the eigenvectors as columns, and a diagonal matric containing the eigenvalues on the main diagonal.

```
# Note: in the output matrix, eigenvectors/eigenvalues are not necessarily ordered by eigenvalue magnitude. You should sort them first.
from numpy import linalg as LA


w, v = LA.eig(cov_bothclass)                              # w : eigenvalues ~a 200 element list   # v : eigen vectors matrix  ~220*220
w = np.array([w])                                         # convert w from a list to an array to be concatenatable  ~1*220

# Sorting
v_sorting = np.concatenate((v,w) , axis = 0)              # append w to v (as the last row) to sort it based on the values of w  ~221*220
v_sorting_df = pd.DataFrame(v_sorting)                                                    # convert array to a dataframe to be sortable
v_sorting_df_sorted = v_sorting_df.sort_values(by = 220, axis = 1 , ascending= False)     # sort columns  descending based on the values of w (last r
v_sorting_sorted = v_sorting_df_sorted.to_numpy()                                         # Convert back to an array
v_sorted = v_sorting_sorted[:220,:]                                                       # Removing w from the last row of v   ~220*220
```

```
14
```

Choose a number of dimensions K<=220.Construct the eigenvector matrix W for K components (i.e., select the last K columns)

```
1 # Considering k first eigenvectors
2
3 k = 50
4 v_new = v_sorted[:,:k]           # ~ 220 * k
5 v_new_transpose = v_new.T        # ~ k  * 220
6
```

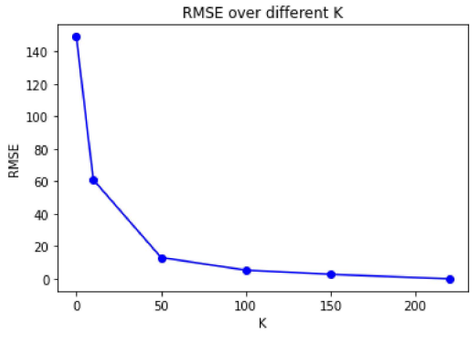
Using W, compute the PCA coefficients for each spectral vector in the data set. Then from the PCA coefficients obtain an approximation of the corresponding vector and compute the error (mean square error - MSE)

```
1 # Note: remember to remove the mean from the vectors of the dataset
2
3
4 z = v_new_transpose @ bothclass_test      # z = w.T @ x_test      # ~ k * 1000
5
6 ##########################
7 bothclass_hat = v_new @ z            # Calculating z_hat
8
9 ##############################
10 # Calculating MSE
11 SE = 0                                # square error
12 for i in range (bothclass_hat.shape[1]):
13     a = np.subtract(bothclass_hat[:,i], bothclass_test[:,i])
14     SE = a @ a.T
15     SE +=SE
16 MSE = SE/ bothclass_hat.shape[1]
17
18 print('The MSE is equal to:',MSE)
19
20
```

```
The MSE is equal to: 171.16245422275685
```

```
1   # plot MSE over K for k = 10 , 50 , 70 , 100
2   import matplotlib.pyplot as plt
3
4
5   def RMSE (a):
6       k = a
7       v_new = v_sorted[:,:k]                    # ~ 220 * k
8       v_new_transpose = v_new.T                 # ~ k  * 220
9       z = v_new_transpose @ bothclass_test      # z = w.T @ x_test      # ~ k * 1000
10      bothclass_hat = v_new @ z                 # Calculating z_hat
11
12      SE = 0                                    # square error
13      for i in range (bothclass_hat.shape[1]):
14          a = np.subtract(bothclass_hat[:,i], bothclass_test[:,i])
15          SE = a.T @ a
16          SE +=SE
17      MSE = SE/ bothclass_hat.shape[1]
18      RMSE = MSE**0.5
19      return RMSE
20
21  a = RMSE(0)
22  b = RMSE(10)
23  c = RMSE(50)
24  d = RMSE(100)
25  e = RMSE(150)
26  f = RMSE(220)
27
28  RMSE_list = [a,b,c,d,e,f]
29  K_list = [0,10,50,100,150,220]
30
31
32  plt.plot(K_list,RMSE_list,'-o', color = 'blue')
33  plt.title("RMSE over different K")
34  plt.ylabel('RMSE')
35  plt.xlabel('K')
36
37
```

```
Text(0.5, 0, 'K')
```

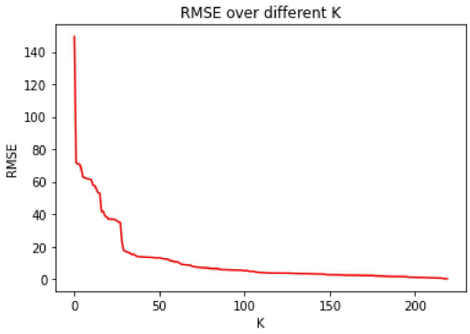


Plot the average MSE over the test set as a function of K.

```
1   import matplotlib.pyplot as plt
2
3   #Insert code here
4
5   def RMSE_over_K (a):
6       k = a
7       v_new = v_sorted[:,:k]                      # Slice the K first Eigen values
```

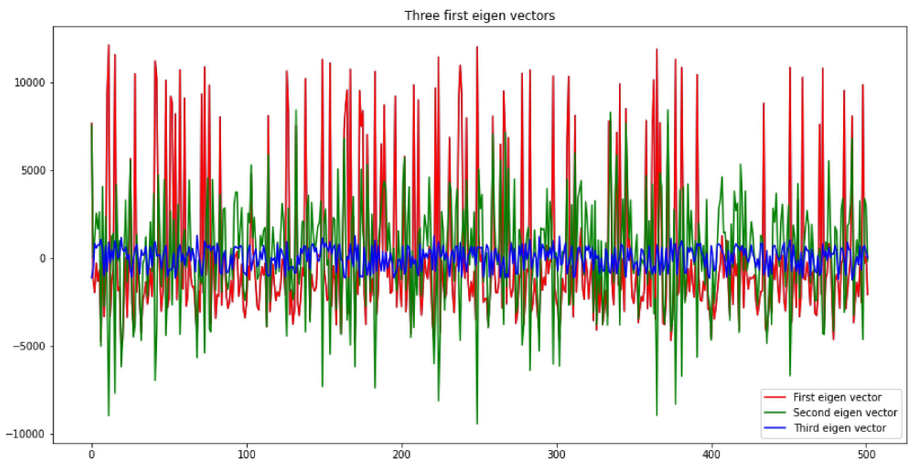
```
    v_new_transpose = v_new.T
    z = v_new_transpose @ bothclass_test
    bothclass_hat = v_new @ z
    SE = 0
    for i in range (bothclass_hat.shape[1]):
        a = np.subtract(bothclass_hat[:,i], bothclass_test[:,i])
        SE = a.T @ a
        SE +=SE
    MSE = SE/bothclass_hat.shape[1]
    RMSE = MSE ** 0.5
    return RMSE

# Create a vector of MSE based on K
RMSE_k = []
for j in range(len(bothclass_hat)):
    RMSE_k.append(RMSE_over_K(j))
print(len(RMSE_k),RMSE_k)
k = np.arange(0,220,1)

plt.plot(k, RMSE_k, color="red")
plt.title("RMSE over different K")
plt.ylabel('RMSE')
plt.xlabel('K')
plt.show()
```

```
220 [149.23858863352567, 71.73053702968461, 70.8877977661219, 70.88373667591365, 68.14855007021173,
```



Plot the eigenvectors corresponding to the 3 largest eigenvalues – this will give you an idea of the basis functions employed by PCA

```
#Insert code here
a = plt.figure(figsize=(15,7.5))
first_eigenvector = z[0,:]
second_eigenvector = z[1,:]
third_eigenvector = z[2,:]

plt.plot(first_eigenvector)
plt.plot(first_eigenvector, color ="red" , label ='First eigen vector')
plt.plot(second_eigenvector, color ="green", label ='Second eigen vector')
plt.plot(third_eigenvector, color ="blue", label ='Third eigen vector')
plt.title('Three first eigen vectors')
plt.legend()
plt.show()
```



Student's comments to Exercise 1

- The comments related to the codes are embdedd into the code boxes
- The first 2 curvy plots clearly shows the high amount of the data carried by the first components and it is clear that by adding more component the reconstruction error tends to get lower and lower; hence it is possible to remove many of the latent variables without loosing much of the data.
- In total, we can infer the following from the plot of RMSE by k : Dimensionality reduction quality - Explanation of Data variance - Data compression quality
- The last figure clearly depicts that the highest variance value is for the first pricipal component the second for the second one and ,.....

Exercise 2 – Classification using dimensionality reduction and whitening (optional)

In this exercise you will apply a simple 2-class linear discriminant analysis (LDA) classifier to the data belonging to the two classes, before and after **whitening** (i.e. applying PCA plus rescaling each coefficient to unit variance: $y = \Lambda^{-\frac{1}{2}} W^T x$. Note that matrix Λ is obtained as one of the

outputs of *eig.m*). This classifier makes the Naïve Bayes Classifier assumption that the features are statistically independent, thus the shared covariance matrix is taken as $\Sigma = I$. We also assume that class 0 and class 1 are equiprobable. Thus, letting μ_0 and μ_1 be the mean of vectors in class 0 and 1, we define $x_0 = 0.5(\mu_0 + \mu_1)$ and $w = \mu_1 - \mu_0$. A test vector x is classified into class 1 or 0 depending on whether $sign(w^T(x - x_0))$ is equal to +1 or -1.

When the classifier is applied to the PCA coefficients, you simply replace μ_0 and μ_1 with their reduced versions (subtracting μ and applying the same PCA matrix computed over the training set), and recalculate x_0 and w accordingly.

Task: Divide the Indian Pines dataset into training and test sets (e.g. 75% of the data of each class to be used as training data, and 25% as test data). Train this classifier on the training data, and apply it to the test data. In particular, you should perform the following:

- Plot the mean vector of class 0 and 1 – this will give you a visual description of the differences among vectors of either class.
- Apply the classifier to the original data (without PCA) and compute its accuracy
- Apply the classifier to the PCA coefficients for different values of K, and compute its accuracy
- Apply the classifier to the original data where only the first K features have been retained, and compute its accuracy (this is a more brutal way to reduce dimensionality)
- Try to classify the data using a Support Vector Machine and compare the results

Plot the mean vector of class 0 and 1 – this will give you a visual description of the differences among vectors of either class.

```
#Insert code here
```

Apply the classifier to the original data (without PCA) and compute its accuracy

```
#For this section you can use the code you developed for the previous Lab or the sklearn function LinearDiscriminantAnalysis
```

Apply the classifier to the PCA coefficients for different values of K, and compute its accuracy

```
#Insert code Here
```

Apply the classifier to the original data where only a subset of K features (selected randomly) have been retained, and compute its accuracy (this is a more brutal way to reduce dimensionality)

```
#Insert code here
```

Try to classify the data using a Support Vector Machine and compare the results

```
#For this section you can use the sklearn library

#from sklearn import svm

#svc = svm.SVC()
#svc.fit(Train, Train_Label)
#predict = svc.predict(Test)

#print(metrics.accuracy_score(predict, Test_Label))
```

Student's comments to Exercise 2

Add comments to the results of Exercise 2 here (may use LateX for formulas if needed).

STATISTICAL LEARNING AND NEURAL NETWORKS, A.A. 2022/2023

COMPUTER LAB 4 - Kalman filter

Duration: 3 hours

Introduction:

In this lab, you are provided with the set of coordinates (x,y – horizontal and vertical) describing the trajectories of pedestrians moving acrooss a scene. Your task is to simulate the observed positions of the pedestrians by adding observation noise, then to track the subjects using a Kalman filter. In other words, you need to estimate the next (x,y) positions, from the observations of the previous positions.

Simulating the observed coordinates

Choose one of the trajectories in the dataset. This data will be considered the real trajectory. Generate the observed directory by adding observation noise $\delta_t \sim \mathcal{N}(0, \sigma_R^2)$ to the (x,y) coordinates.

Designing the Kalman filter

Your task is to **design a Kalman filter** based on a constant velocity model, which tracks the next (x,y) position of the object, from the observation of the previous positions. The code must be based on the following model.

- The state vector contains coordinates and velocities: $z_t^T = (z_{1t}, z_{2t}, v_{1t}, v_{2t})$ (see slides). The object has initial coordinates (0,0) and velocity (Δ, Δ).
- Only the coordinates (but not the velocities) are observed. This leads to a linear dynamical system with:

$$A = \begin{pmatrix} 1 & 0 & \Delta & 0 \\ 0 & 1 & 0 & \Delta \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$C = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

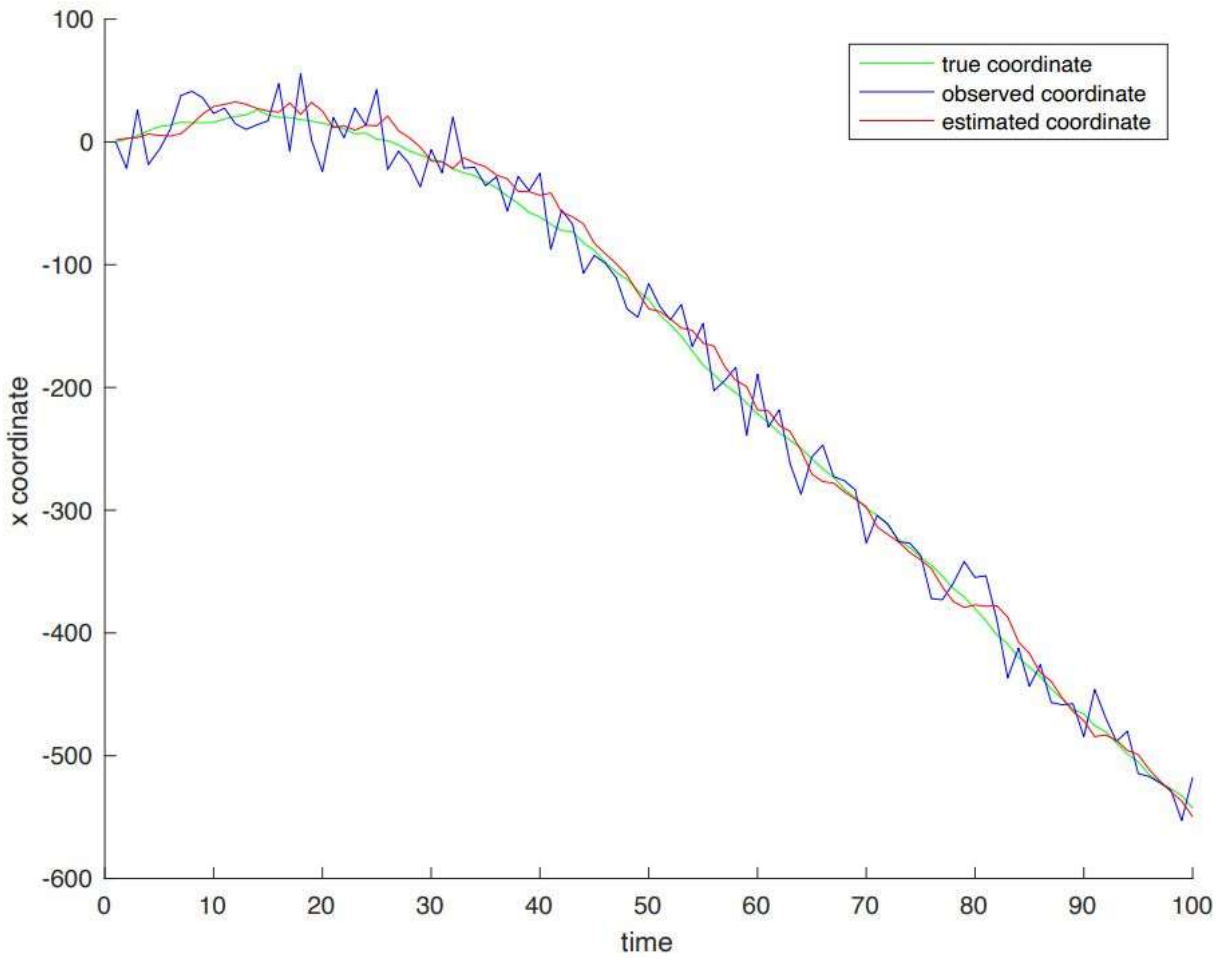
and B = D = 0.

- Σ_Q, and Σ_R should be set to:

$$\Sigma_Q = \sigma_Q^2 \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\Sigma_R = \sigma_R^2 \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

To see if your Kalman filter is working well, you should plot the estimated position of the object over time with respect to the true position (i.e., the first two entries of the state vector) and the observed position. Depending on the chosen parameters, for each coordinate the graph may look something like this:



Suggestion: when implementing your Kalman filter, you will have to choose initial values for μ_t and Σ_t. Provided that you do not make very unreasonable assumptions, the Kalman filter will update those estimates from observed data, so the initial choices are not very critical.

Test your Kalman filter modifying the values of some of the parameters, including standard deviations σ_Q and σ_R, initial values for μ_t and Σ_t and the value of Δ.

New section

```
from google.colab import drive
drive.mount('/content/drive')

!pip install ndjson
```

```
Mounted at /content/drive
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting ndjson
  Downloading ndjson-0.3.1-py2.py3-none-any.whl (5.3 kB)
Installing collected packages: ndjson
Successfully installed ndjson-0.3.1
```

```
import ndjson
import pandas as pd
import numpy as np

#Dataset origin: https://paperswithcode.com/dataset/trajnet-1
#The crowds_students001 file is loaded, and formatted as a list of numpy vectors

with open('/content/drive/MyDrive/Labs/Lab4/crowds_students001_trackonly.ndjson') as f:
  data = ndjson.load(f)

```

```
p='-1'

person_dict = []

for ii in range(len(data)):
  if(p!=data[ii]['track']['p']):
    p=data[ii]['track']['p']
    person_dict.append([])
  person_dict[p].append((data[ii]['track']['x'],data[ii]['track']['y']))

person_dict_numpy = []
for ii in range(148):
  person_dict_numpy.append(np.array(person_dict[ii], dtype=float))

print(data[1]['track']['p'])
print('data : ' , len(data) , data)
print('person_dict 68: ' , person_dict[68])
print('person_dict_numpy : ' , len(person_dict_numpy))
```

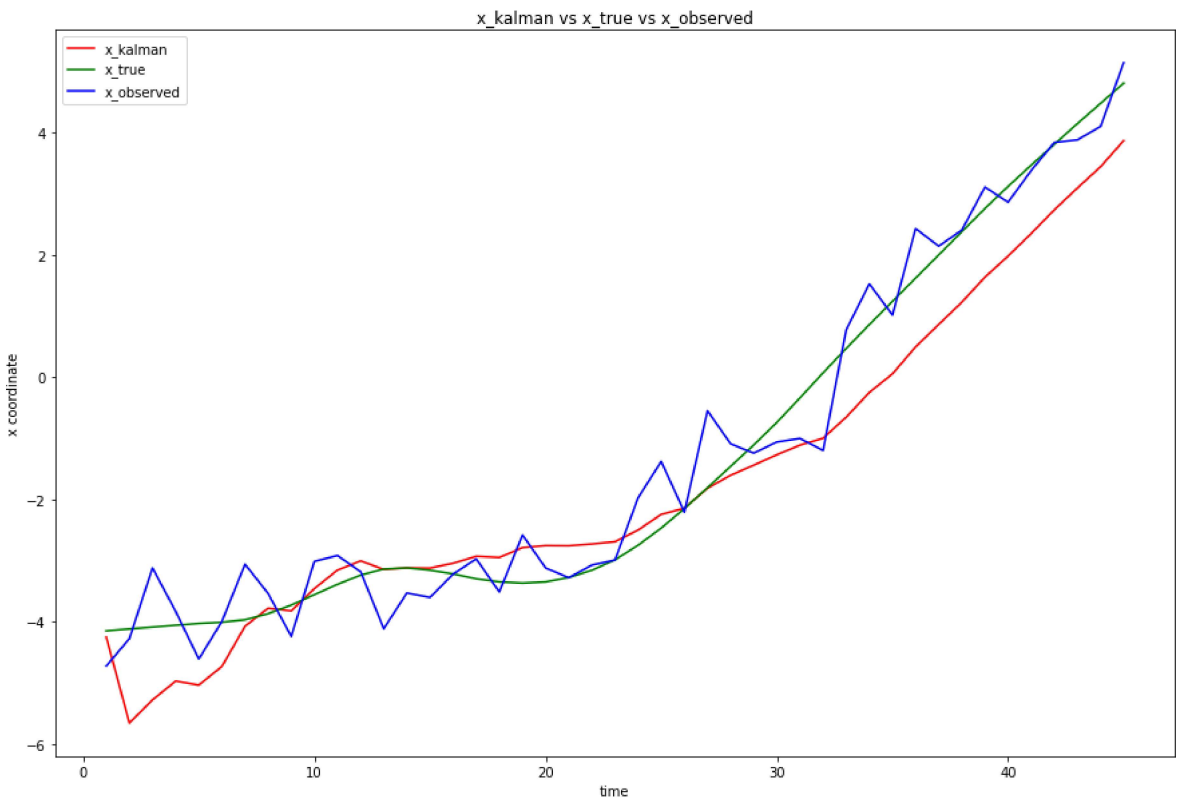
```
0
data :  21335 [{'track': {'f': 10, 'p': 0, 'x': 3.21, 'y': -3.21}}, {'track': {'f': 20, 'p': 0, 'x': 2.76, 'y': -3.28}}, {'track': {'f': 30, 'p': 0, 'x': 2.3, 'y': -3.34}}, {'track': {'f': 40, 'p':
person_dict 68:  [(-4.17, 6.8), (-4.15, 6.54), (-4.12, 6.31), (-4.09, 6.09), (-4.06, 5.89), (-4.03, 5.71), (-4.01, 5.54), (-3.97, 5.39), (-3.87, 5.27), (-3.73, 5.16), (-3.56, 5.05), (-3.39, 4.93),
person_dict_numpy :  148
```

```
import numpy as np
import matplotlib.pyplot as plt

# Implement the kalman filter as a function using the numpy library, the matrix products can be done using the np.dot function
# The matrix inversion can be done using the np.linalg.inv

def kalman_filter(y, mu_tmin1, sigma_tmin1, A, C, Q, R):

  # Predict the state at the next time step
    mu_t_tmin1 = np.dot(A, mu_tmin1)    # (4,4)*(4,1) = (4,1)
    sigma_t_tmin1 = np.dot(np.dot(A, sigma_tmin1), A.T) + Q  # (4*4)
    y_hat = np.dot(C, mu_tmin1)  # (2*4)(4*1)  = (2*1)

  # Calculate the Kalman gain
    S = R + np.dot(C, np.dot(sigma_t_tmin1, C.T)) # (2,2) + (2,4)*(4,4)*(4,2) = (2,2)
    k_gain = np.dot(np.dot(sigma_t_tmin1, C.T), np.linalg.inv(S)) # (4,4)(4*2)(2*2) = (4*2)


  # Update the estimate of the state
    y = np.array([y]).T
    r = np.subtract(y,y_hat) # (2*1)

    mu_est = mu_t_tmin1 + np.dot (k_gain , r) # (4,1) + (4,2)(2,1) = (4,1)
    I = np.identity(4)
    sigma_est = np.dot((I - np.dot(k_gain, C)), sigma_t_tmin1) # ((4,4)- (4,4)(4,4))(4,4) = (4,4)

    return mu_est, sigma_est



true_positions = person_dict_numpy[68] #choose a single trajectory by taking an element of the list person_dict_numpy, select a random index between 0 and 148

# Define parameter delta
Delta = 1 # constant velocity

A = np.array([[1, 0, Delta, 0],
              [0, 1, 0, Delta],
              [0, 0, 1, 0 ],
              [0, 0, 0, 1 ]])


# Define the measurement matrix
C = np.array([[1, 0, 0, 0],  # x is measured directly
              [0, 1, 0, 0]]) # y is measured directly

# Set the standard deviation of the measurement noise

sigma_R = 0.5
sigma_Q = 0.003

# Define the process noise covariance matrix
Q = (sigma_Q**2)*np.eye(4)

# Define the measurement noise covariance matrix
R = (sigma_R**2)*np.eye(2)

# Set the initial state and covariance
mu_0 = [0.1,0.03,0.1,0.9]
mu_0 = np.array([mu_0])
mu_0 = mu_0.T
sigma_0 = ([[1, 0, 1, 0],
            [0, 1, 0, 1],
            [0, 0, 1, 0 ],
            [0, 0, 0, 1 ]])
# Iterate over the observed coordinates
y_est = []
y_est.append((mu_0,sigma_0))

t_series = []
x_kalman = []
y_kalman = []
x_observed = []
y_observed = []
x_true = []
y_true = []


for t in range(1,len(true_positions)):
  # Get the observed coordinates at time t
  # Note: the observed position is simulated by adding gaussian noise to the true_positions
  y = true_positions[t] + np.random.normal(0,sigma_R,2)
  #apply the kalman filter on the observed coordinates

  y_est.append(kalman_filter(y, y_est[t-1][0], y_est[t-1][1], A, C, Q, R))
  x_kalman.append(y_est[t][0][0].tolist())    # Creating x series
  y_kalman.append(y_est[t][0][1].tolist())    # Creating y series

  t_series.append(t)                  # Creating a time series for plotting

  x_true.append(true_positions[t][0])
  y_true.append(true_positions[t][1])

  x_observed.append(y[0])
  y_observed.append(y[1])


print()

print('t_series :',t_series)
print('\n x_kalman :',x_kalman)
print('\n y_kalman :',y_kalman)
print('\n x_true :', x_true)
```

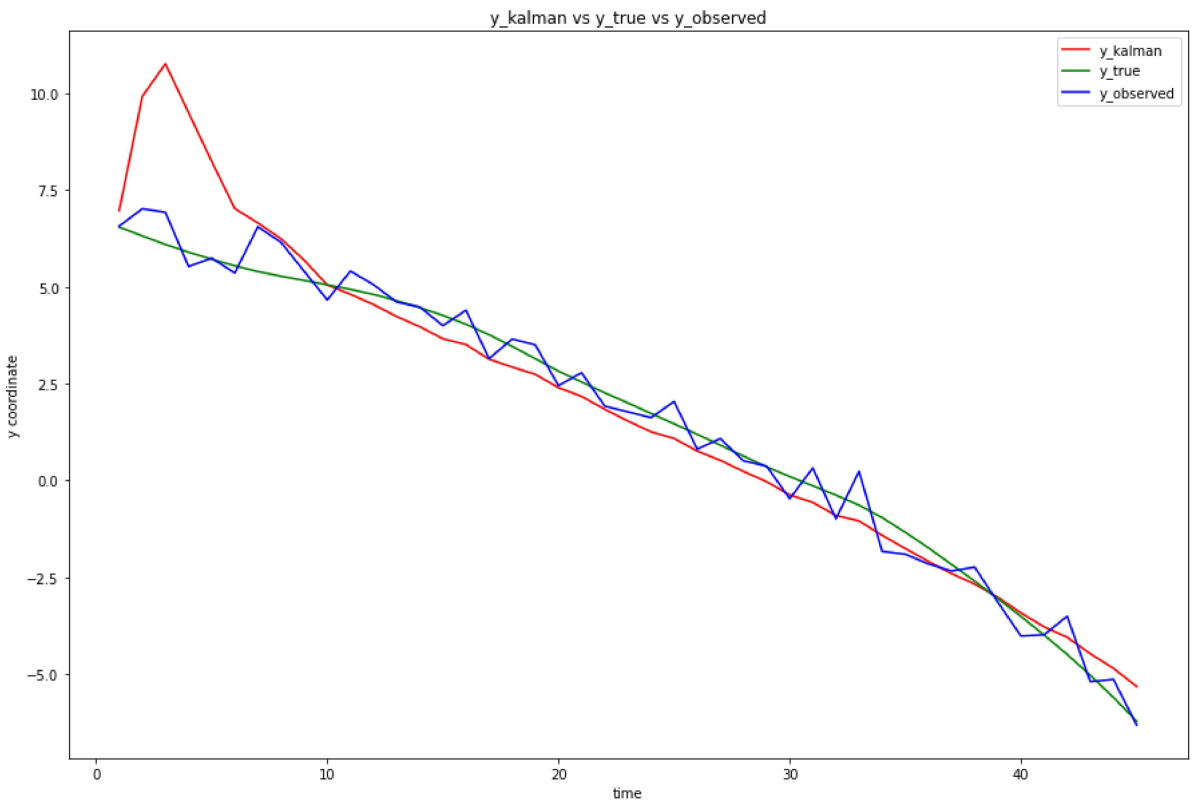
```
print('\n y_true :', y_true)
print('\n x_observed :', x_observed)
print('\n y_observed :', y_observed)


#Use the matplotlib library to plot the true positions, observed positions y and the results of the kalman filtering
#You should obtain a plot which resembles the one in the figure
#Plot the trajectory of the x coordinate over time and the trajectory of the y coordinate over time into two separate plot

```

```
t_series : [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45]

 x_kalman : [[-4.255485475408207], [-5.658536659373023], [-5.281772407415517], [-4.973906366166316], [-5.039233049046499], [-4.734219723440784], [-4.076267638507], [-3.780997340821777], [-3.825637

 y_kalman : [[6.960592743462526], [9.909720986350843], [10.75381601625452], [9.493094513759166], [8.23563598804174], [7.016455056104876], [6.645319751710337], [6.2370244878535965], [5.6852974810319

 x_true : [-4.15, -4.12, -4.09, -4.06, -4.03, -4.01, -3.97, -3.87, -3.73, -3.56, -3.39, -3.24, -3.14, -3.12, -3.16, -3.22, -3.3, -3.35, -3.37, -3.35, -3.28, -3.16, -2.99, -2.75, -2.47, -2.15, -1.81

 y_true : [6.54, 6.31, 6.09, 5.89, 5.71, 5.54, 5.39, 5.27, 5.16, 5.05, 4.93, 4.8, 4.64, 4.46, 4.26, 4.03, 3.76, 3.46, 3.14, 2.82, 2.54, 2.26, 2.0, 1.73, 1.46, 1.19, 0.91, 0.63, 0.35, 0.1, -0.14, -0

 x_observed : [-4.726774817824198, -4.278665979030187, -3.1218432666372156, -3.833635430463483, -4.613177373336612, -3.9971115367479637, -3.0612939494674256, -3.540978375919125, -4.24210179051258,

 y_observed : [6.563140631107409, 7.011033198578276, 6.921075995955592, 5.52038121249522, 5.737102115256241, 5.354001020363288, 6.545660925692861, 6.144584680217733, 5.3976227925041345, 4.658056620
```

```
# Plotting x
f = plt.figure()
f.set_figwidth(15)
f.set_figheight(10)
plt.plot(t_series, x_kalman , color = 'red')
plt.plot(t_series, x_true , color = 'green')
plt.plot(t_series, x_observed , color = 'blue')
plt.legend(['x_kalman', 'x_true', 'x_observed'])
plt.title('x_kalman vs x_true vs x_observed')
plt.xlabel ('time')
plt.ylabel('x coordinate')
plt.show()


```




```
# Plotting y
f = plt.figure()
f.set_figwidth(15)
f.set_figheight(10)
plt.plot(t_series, y_kalman , color = 'red')
plt.plot(t_series, y_true , color = 'green')
plt.plot(t_series, y_observed , color = 'blue')
plt.legend(['y_kalman', 'y_true', 'y_observed'])
plt.title('y_kalman vs y_true vs y_observed')
plt.xlabel ('time')
plt.ylabel('y coordinate')
plt.show()
```



Student's comments to COMPUTER LAB 4 - Kalman filter

- Kalman filter is used to find/predict the optimal estimate of a system when we can not directly measure or estimate it.
- It does it by combining the 1-measurement/observation(y) and 2-prediction(z) in two different phases : I.predict II.update

- We can make kalman filter follow the measurement or the model by:
    1. Tuning measurement noise: The measurement noise is the uncertainty associated with the measurement, and it is used to determine the weight assigned to the measurement update. By decreasing the measurement noise, you increase the weight assigned to the measurement, and make the Kalman estimate follow the measured value more closely
    2. Tuning process/model noise: The process noise is the uncertainty associated with the model, and it is used to determine the weight assigned to the model prediction. By decreasing the process noise, you increase the weight assigned to the model, and make the Kalman estimate trust the model more.
- We choose the delta = 1 since it is a constant velocity model

NOTE for figure:

- The Kalman filter may not have precise estimation at the beginning due to uncertainties in the initial conditions, model, and measurements, and **it may take some time for the estimate to converge to the true state of the system**