Machine Learning Programming
Assignment 3: K-Nearest Neighbors

Professor: Aude Billard
Assistants: Nadia Figueroa, Laila El Hamamsy,
Hala Khodr and Lukas Huber
Contacts:
aude.billard@epfl.ch, nadia.figueroafernandez@epfl.ch
laila.elhamamsy@epfl.ch, hala.khodr@epfl.ch, lukas.huber@epfl.ch

Winter Semester 2017

Introduction

In this practical, you will code the K-Nearest Neighbors algorithm in Matlab. You will then use the code for classification on 2D datasets before applying it on higher dimensional datasets.

Submission Instructions

Deadline: November 14, 2017 @ 6pm. Assignments must be turned in by the deadline. 1pt will be removed for each day late. A day late starts one hour after the deadline.
Procedure: From the course Moodle webpage, the student should download and extract the .zip file named `TP3-KNN-Assignment.zip` which contains the following files:

Part 1 - Algorithm	Part 2 - Applications
split_data.m	confusion_matrix
my_knn.m	knn_ROC.m
my_accuracy.m	cross_validation.m
plot_knn_acc.m	--
test_knn_2d.m	test_knn_bcd.m

As well as `TP3-KNN-Dataset.zip` which contains the datasets required to test your functions.

Assignment Instructions

The assignment consists on implementing the blue colored MATLAB functions from scratch. These functions can be tested with the `test_*.m`. These testing scripts depend on `ML_toolbox`, which must be downloaded from: `https://github.com/epfl-lasa/ML_toolbox`. Before proceeding make sure that all the sub-directories of the ML_toolbox have been added to your MATLAB search path. This can be done as follows in the MATLAB command window:

```
>> addpath(genpath('path_to_ML_toolbox'))
```

Once you have tested your functions, you can submit them as a .zip file with the name: `TP3-KNN-Assignment-SCIPER.zip` on the submission link in the Moodle webpage. Your submission archive should contain ONLY the following:

```
TP3-KNN-Assignment-SCIPER
└── Part1
```

Part2

DO NOT upload `ML_toolbox`, the `check_utils` directory or the `Datasets` directory.

1 Part 1: K-Nearest Neighbors (K-NN)

K-NN (K-Nearest Neighbors) is a nonparametric "lazy" learning algorithm that can be used for classification or regression. In this assignment we will cover its use for classification purposes only. It is one of the simplest classification algorithms in the machine learning literature, generally used as a baseline for more complex algorithms.

K-NN is considered as *nonparametric* as it does not make any theoretical assumptions of the distribution of the underlying data (e.g. linearly separable, normally distributed, etc.). It's considered a *lazy* algorithm because it does not learn any parameters or model to generalize the observed data, it rather uses the full training dataset to find the best outcome for a query point. This is done by keeping the complete training data during testing and using a distance-based majority vote mechanism, to predict a label for a new sample (i.e. query point).

K-NN is a simpler version of DBSCAN presented in the Applied Machine Learning course for supervised learning tasks (i.e. classification). Like DBSCAN, it groups points according to how distant they are from each other using a minimum (K) of points. DBSCAN adds a condition on the minimal distance to detect outliers and merge the clusters. K-NN does not have these two additional mechanisms and is hence very sensitive to outliers.

K-Nearest Neighbor Algorithm

In classification problems we are given a training dataset $D = \{(\mathbf{x}^1, y^1), (\mathbf{x}^2, y^2), \ldots, (\mathbf{x}^M, y^M)\}$ where $\mathbf{x}^i \in \mathbb{R}^N$ is the i-th N-dimensional data point (or feature vector) for $i = 1, \ldots, M$ and $y^i \in \{0, 1\}$ is the categorical outcome (or class label) corresponding to each data point. The goal is then, given a new sample (or query point) $\mathbf{x}' \in \mathbb{R}^N$ we would like to predict its label $\hat{y}' \in \{0, 1\}$[1]. The K-NN algorithm addresses this problem by applying a very simple rule (For further reading, refer to Chapter 4 of the Pattern Classification book by Duda et al. [1]:

> *"$\mathbf{x}'$ is assigned the label $\hat{y}'$ most frequently represented among its k nearest samples."*

Hence, the classification decision for $\mathbf{x}'$ involves the following steps:

1. Calculate the pairwise distances between $\mathbf{x}'$ and all points in the training dataset $D_x = \{\mathbf{x}^1, \ldots, \mathbf{x}^M\}$.
2. Extract the k nearest neighbors of $\mathbf{x}'$ from the pairwise distances computed in step 1 $\mathbf{x}'_k = \{x_1, \ldots, x_k\}$ and their corresponding class labels $\mathbf{y}^k = [y^1, \ldots, y^k]$.
3. Majority vote on class labels based on the k nearest neighbor list $\mathbf{y}^k$.

In other words, the K-NN algorithm begins with a query point $\mathbf{x}'$ (as can be seen in Figure 1) and grows a spherical region until it encloses k training points, regardless of their labels. The query point is then labeled by a majority vote from the enclosed training points.

[1] We use $\hat{\cdot}$ for $\hat{y}$ to indicate that it is an estimated value and not the true label y.

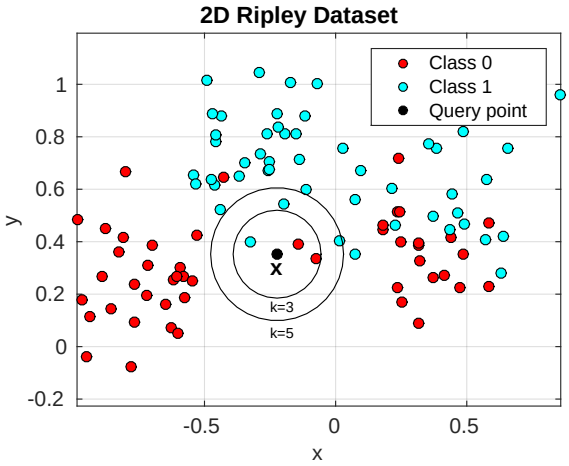

Figure 1: KNN example on the Ripley Dataset (generated by a mixture of two Gaussian distributions). In the case of $k = 3$, $\mathbf{x}$ would be labeled as class 0. However, in the case of $k = 5$, $\mathbf{x}$ would be labeled as class 1.

Special considerations of KNN:

- k must be an odd value for an even-class problem. This avoids ties during the majority voting step. The inverse holds, use an even number for k when you have an odd-class problem.
- Choosing k is the most important step in this algorithm. If $k \to 1$ is too small, noise or outliers in the data will have a higher effect on the result. On the other hand, if $k \to M$ is too large, computation time is hindered since KNN has a computational complexity of $\mathcal{O}(Mk)$. Moreover, setting k to a large value, contradicts the idea behind KNN; i.e. points closer together belong to the same class. Intuitively, one can assume equal distribution of points across classes and we can find a feasible range of k around $k = \kappa * M/C$ with $0 < \kappa <= 1$ and C is the number of classes, which is the ratio of number of datapoints and number of classes. This is not a strict rule, but can be useful when finding an optimal k, one can also make an informed decision of k by analyzing the density of the points, balance in classes, etc.

1.1 Data Handling for Classification

We will test our implementation of KNN on the 2D concentric circle dataset shown in Figure 2. For any type of classification algorithm, one must first split the dataset into *training* and *testing* sets. This train/test split is used in order to reduce overfitting with your classifier. By using the full dataset for training, the model or classifier will tend to overfit to the noise of the observed data, rather than the real, underlying model. The data points in your *training* set are used to tune parameters (i.e. choose the best k for KNN) or learn models of your classifier. The data points on your *testing* set are considered as a separate dataset used solely to evaluate the performance of your learned classifier.
To partition a dataset for classification one generally selects a Training/Testing ratio: `tt_ratio`, where `tt_ratio`=0.6 corresponds to 60% of your points used for *training* and 40% of your points used for *testing*. One can vary the `tt_ratio` to have a better estimate of the classifier's performance. In general, `tt_ratios` are tested in the range of $\{70/30, 60/40, 50/50, 40/60, 30/70\}$, depending on how much data you have. Special considerations of train/test sets:

1. A large test set generally gives a more reliable estimate of the classifier's accuracy (i.e. a lower variance estimate).
2. A large training set will generally be more representative of how much data we actually have for learning process.

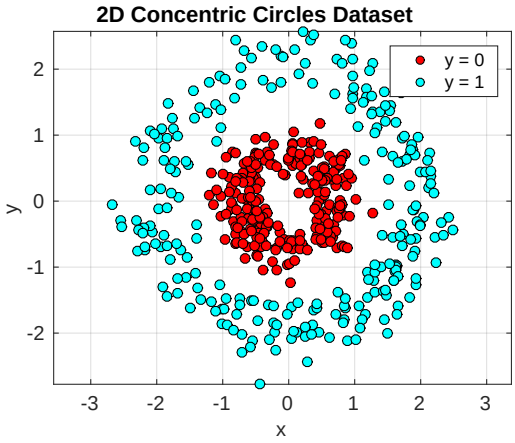

Figure 2: 2D Concentric circles Dataset. $M = 500$, $N = 2$, $y \in 0, 1$

3. Using a single training set does not give a reliable accuracy estimate. It cannot tell us how sensitive the classifier is wrt. specific samples.

Following these considerations, we will implement a `split_data.m` function which should:

1. Randomize the indices of the datapoints, so that every time the function is called it gives you a new set of points.
2. Extract the first subset of indices as the *train* set and the remaining as the *test* set.
3. Output $\mathbf{X_{train}}, \mathbf{y_{train}}$ and $\mathbf{X_{test}}, \mathbf{y_{test}}$.

TASK 1: Implement split_data.m function (2pts)

```
function [ X_train, y_train, X_test, y_test ] =  split_data(X, y, tt_ratio)
%SPLIT_DATA Randomly partitions a dataset into train/test sets using
%   according to the given tt_ratio
%
%   input ------------------------------------------------------------------
%
%       o X        : (N x M), a data set with M samples each being of
%                    dimension N each column corresponds to a datapoint
%       o y        : (1 x M), a vector with labels y \in {0,1} corresponding
%                    to X.
%       o tt_ratio : train/test ratio.
%   output -----------------------------------------------------------------
%
%       o X_train  : (N x M_train), a data set with M_test samples each being
%                    of dimension N.
%                            each column corresponds to a datapoint
%       o y_train  : (1 x M_train), a vector with labels y \in {0,1}
%                    corresponding to X_train.
%       o X_test   : (N x M_test), a data set with M_test samples each being
%                    of dimension N each column corresponds to a datapoint
%       o y_test   : (1 x M_test), a vector with labels y \in {0,1}
%                    corresponding to X_test.
```

`Implementation Hint:` Useful function `randperm()`.

Test Implementation

To test that we have implemented the `split_data.m` function correctly, we can run the **first two** code blocks in the MATLAB script `test_knn_2d.m`. This will load a 2D dataset shown in

Figure 2 (by selecting only the first sub-block `1a) Load 2D Concentric Circles`) and if your function is correct you will see a plot *similar* to Figure 3, by changing `tt_ratio = 0.3` you should visualize a plot *similar* to Figure 4. Besides visual inspection, the **second** code block has an encrypted `test_splitdata.p` function, which evaluates your data splits. If you get the following messages on your MATLAB command window, you can move on to the next task.

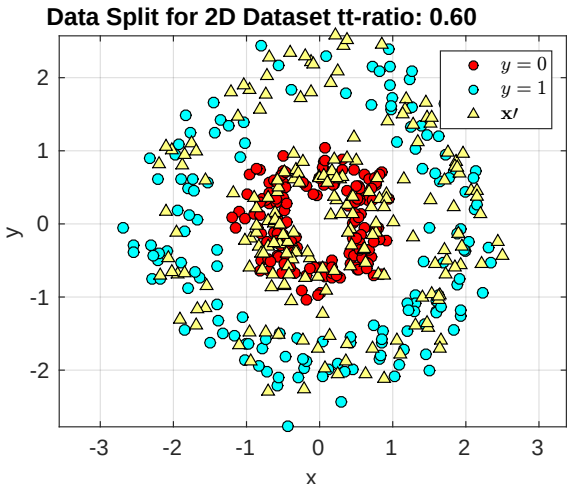

Figure 3: 2D Concentric Dataset with $tt_{ratio} = 0.6$, $M_{train} = 300$ and $M_{test} = 200$.

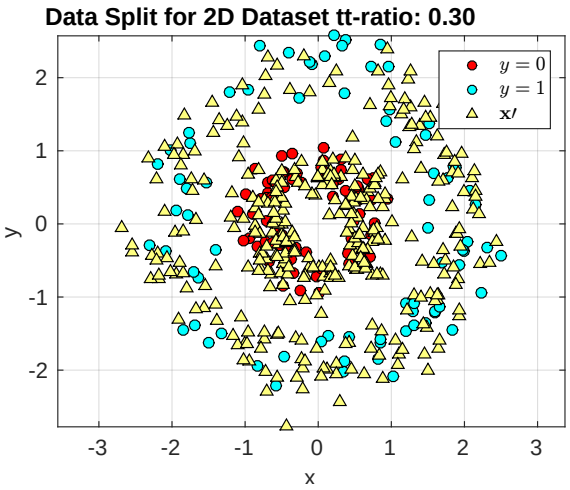

Figure 4: 2D Concentric Dataset with $tt_{ratio} = 0.3$, $M_{train} = 150$ and $M_{test} = 350$.

```
--- Testing data_splits.m ---
[Test 1] Number of points in splits: Correct
[Test 2] Train and Test Splits Randomized: Correct
```

1.2 Distances for KNN Classification

The KNN classifier relies on a metric or "distance" function between the data points in order to accomplish the first step of the algorithm:

1. Calculate the pairwise distances between $\mathbf{x}'$ *and all points in the training dataset* $D_x = \{\mathbf{x}^1, \ldots, \mathbf{x}^M\}$.

Such a distance function has already been implemented in the `my_distance.m` function for the `TP2-KMeans-Assignment`. Although we have three types of distance implemented in this function, for simplicity, during this assignment we will use the Euclidean distance:

$$d_{L_2}(\mathbf{x}, \mathbf{x}') = ||\mathbf{x} - \mathbf{x}'||_2 = \sqrt{\sum_{i=1}^{N} |x_i - x'_i|^2}. \tag{1}$$

which should be called as `my_distance(x_1,x_2,'L2')`. For ease of use, we provide the correct implementation of this function in the `./utils` folder.

1.3 KNN algorithm

1.3.1 Compute Pairwise Distances

Now that we have a distance function, we can start implementing the KNN algorithm. The first step is to compute the pairwise distances between $\mathbf{x}'$ and D_x,

$$\mathbf{d} = \{d^i(\mathbf{x}', \mathbf{x}^i) \quad | \quad \forall i = 1, \ldots, M\} \tag{2}$$

where $\mathbf{d} \in \mathbb{R}^M$, is 1

1.3.2 Extract k-Nearest Neighbors

To extract the k-nearest neighbors, one then chooses the k elements of $\mathbf{d}$ which have the smallest distance. This can be done by first sorting the $\mathbf{d}_{(s)}$ in ascending order

$$\mathbf{d}_{(s)} = \{d^i_{(s)} \quad | \quad d^i_{(s)} < d^{i+1}_{(s)} < \cdots < d^M_{(s)}\}$$

and then selecting the subset of k points and labels that are closest to $\mathbf{x}'$:

$$D_{knn} = \{(\mathbf{x}^{I(d^1_{(s)})}, y^{I(d^1_{(s)})}), (\mathbf{x}^{I(d^2_{(s)})}, y^{I(d^2_{(s)})}), \ldots, (\mathbf{x}^{I(d^k_{(s)})}, y^{I(d^k_{(s)})})\}, \tag{3}$$

where $I(d^i_{(s)})$ outputs the index in the original dataset D of the selected point.

1.3.3 Majority Vote

Once $D_{knn} = \{(\mathbf{x}^1_{knn}, y^1_{knn}), \ldots, (\mathbf{x}^k_{knn}, y^k_{knn})\}$ has been retrieved from the previous step, we can estimate the label of $\mathbf{x}'$ by a majority vote from D_{knn}:

$$\hat{y}' = \begin{cases} 0 & \text{if} \quad \sum_{i=1}^k \delta_{(y^i_{knn},0)} > k - \sum_{i=1}^k \delta_{(y^i_{knn},0)} \\ 1 & \text{otherwise} \end{cases}, \tag{4}$$

where

$$\delta_{(y^i_{knn},0)} = \begin{cases} 1 & \text{if} \quad y^i_{knn} = 0 \\ 0 & \text{otherwise} \end{cases}.$$

Now we will implement these three steps in the `my_knn.m` function.

TASK 2: Implement my_knn.m function (6pts)

```
function [y_est] =  my_knn(X_train, y_train, X_test, k, type)
%MY_KNN Implementation of the k-nearest neighbor algorithm
%   for classification.
%
%   input ----------------------------------------------------------------
%
%       o X_train  : (N x M_train), a data set with M_test samples each
%                  of dimension N, each column corresponds to a datapoint
%       o y_train  : (1 x M_train), a vector with labels y \in {0,1}
%                    corresponding to X_train.
%       o X_test   : (N x M_test), a data set with M_test samples each being
%                    of dimension N each column corresponds to a datapoint
%       o k        : number of 'k' nearest neighbors
%       o type     : (string), type of distance {'L1','L2','LInf'}
%
%   output ---------------------------------------------------------------
%
%       o y_est    : (1 x M_test), a vector with estimated labels y \in {0,1}
%                    corresponding to X_test.
```

Implementation Hint: Useful functions: `my_distance()`, `sort()` and `unique()`.

Test Implementation

By running the **third** code block, you will visualize the results of your KNN implementation and the encrypted `test_myknn.p` function will compare the predicted labels from your `my_knn.m`

function with the ones from KNN implemented in `ML_toolbox`. If you get the following messages for the entire range of K on your MATLAB command window, you can move on to the next task.

```
--- Testing my_knn.m with k=1:2:44 ---
[Test k=1] kNN result is: CORRECT
[Test k=3] kNN result is: CORRECT
[Test k=5] kNN result is: CORRECT
[Test k=7] kNN result is: CORRECT
[Test k=9] kNN result is: CORRECT
...
```

1.4 Classification Accuracy

To evaluate the performance of our classifier, we can estimate the classification accuracy. Given the true test labels $\mathbf{y}'$ and the estimated test labels $\hat{\mathbf{y}}'$ for a test set D_{test} of M_{test} points the accuracy can be computed as follows,

$$acc = \frac{\sum_{i=1}^{M_{test}} \delta_{(y_i', \hat{y}_i')}}{M_{test}} \tag{5}$$

where $\delta_{(y_i', \hat{y}_i')} = \begin{cases} 1 & \text{if} \quad y_i' = \hat{y}_i' \\ 0 & \text{otherwise} \end{cases}$. In other words, the accuracy is the percentage of correctly classified data points from all points in D_{test}.

TASK 3: Implement my_accuracy.m function (1pt)

```
function [acc] =  my_accuracy(y_test , y_est)
%My_accuracy Computes the accuracy of a given classification estimate.
%   input ------------------------------------------------------------------
%
%       o y_test   : (1 x M_test),  true labels from testing set
%       o y_est    : (1 x M_test),  estimated labes from testing set
%
%   output -----------------------------------------------------------------
%
%       o acc      : classifier accuracy
```

Test Implementation

By running the **fourth** code block, the encrypted `test_myaccuracy.p` function will compare the estimated accuracy from your `my_accuracy.m` function with the `ML_toolbox` function.

```
--- Testing my_accuracy.m ---
[Test] Accuracy implementation: Correct.
```

Moreover, by running the **fifth** code block, you will be able to visualize the results of your `my_knn.m` and `my_accuracy.m` functions, as in Figure 5 for $k = 1$. You can modify k to have the following values $k = \{1, 5, 15, 35, 61, 75\}$, to see how k-NN behaves on the same dataset. If you obtain the plots in Figure6, 7, 8, 9 and 10, you can move on to the next task. You must take into consideration that since `tt_ratio=0.3`, due to the random data split you can have different values of *accuracy* for the same K. By running the decision-boundary code block, one can visualize the decision boundaries for each output as in Figures 11,12 and 13.

NOTE: Given the random nature of the split_data function, the following plots don't necessarily have to be the same. In fact, for a low `tt_ratio`; i.e. `tt_ratio` < 0.3 and a high K; i.e. $K > 30$ this will undoubtedly be the case.

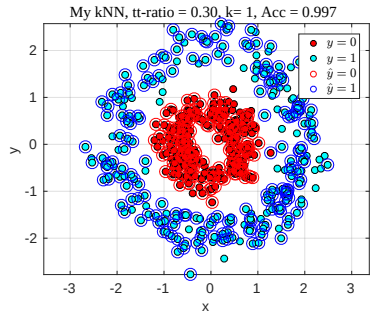

Figure 5: 2D Concentric Dataset k(1)-NN Results.

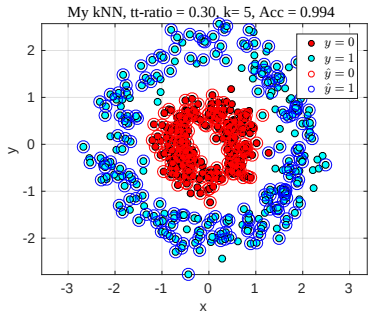

Figure 6: 2D Concentric Dataset k(5)-NN Results.

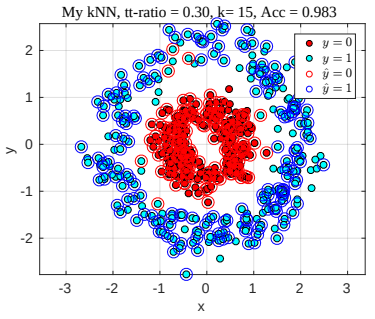

Figure 7: 2D Concentric Dataset k(15)-NN Results.

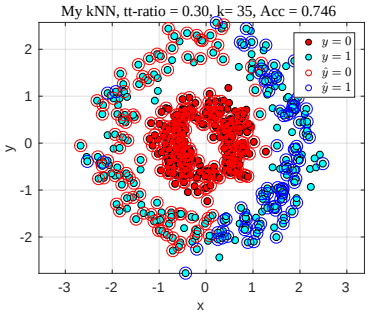

Figure 8: 2D Concentric Dataset k(35)-NN Results.

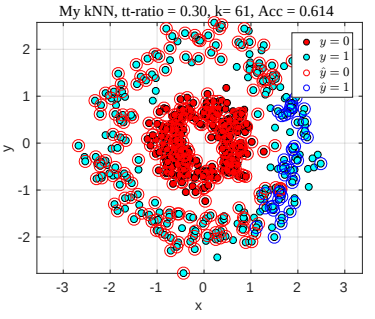

Figure 9: 2D Concentric Dataset k(61)-NN Results.

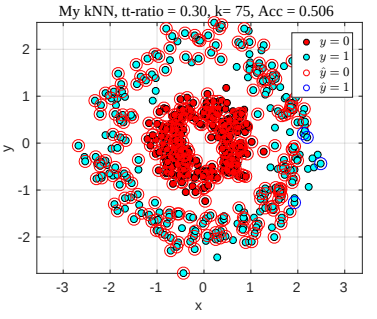

Figure 10: 2D Concentric Dataset k(75)-NN Results.

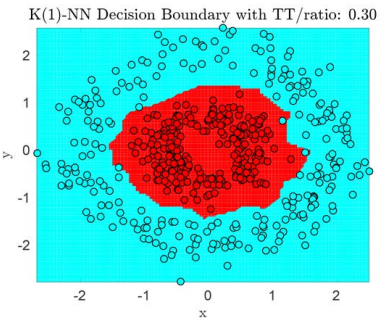

Figure 11: 2D Concentric Dataset k(1)-NN Decision Boundaries.

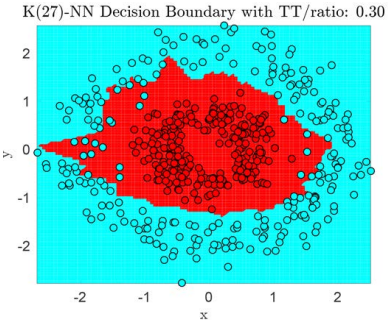

Figure 12: 2D Concentric Dataset k(27)-NN Decision Boundaries.

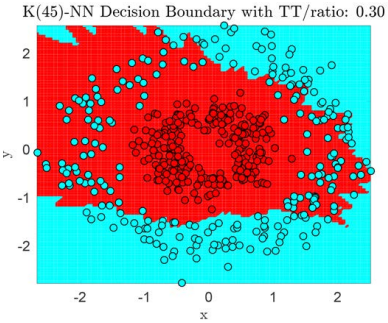

Figure 13: 2D Concentric Dataset k(45)-NN Decision Boundaries.

1.5 Choosing the optimal k for kNN Classification

Until now, we have not addressed the issue of selecting the appropriate k value for our dataset and how it influences the classification result. This can be done by analyzing the *acc* for a range of k values, the same way we analyzed k-means.

TASK 4: Implement knn_eval.m function (1pt)

```
function [acc_curve] = knn_eval( X_train, y_train, X_test, y_test, k_range )
%KNN_EVAL Implementation of kNN evaluation.
%   input --------------------------------------------------------------------
%       o X_train   : (N x M_train), a data set with M_test samples each ...
    being of dimension N.
%                           each column corresponds to a datapoint
%       o y_train   : (1 x M_train), a vector with labels y \in {0,1} ...
    corresponding to X_train.
%       o X_test    : (N x M_test), a data set with M_test samples each ...
    being of dimension N.
%                           each column corresponds to a datapoint
%       o y_test    : (1 x M_test), a vector with labels y \in {0,1} ...
    corresponding to X_test.
%       o k_range   : (1 X K), Range of k-values to evaluate
%
%   output -------------------------------------------------------------------
%       o acc_curve : (1 X K), Accuracy for each value of K
```

Test Implementation

By running the **sixth** code block, you can directly test **knn_eval.m** function and visualize your plots . The output of your function must be the accuracy curve, if implemented correctly you should be able to plot Figure 14 for the 2D-Concentric Circle dataset and Figure 15 for the 9D Breast Cancer Dataset which can be loaded with the MATLAB script `test_knn_9D.m`.

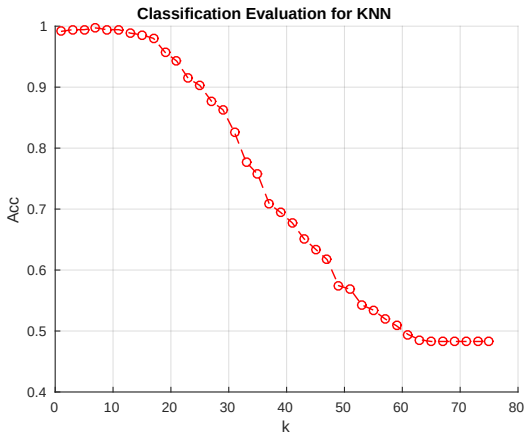

Figure 14: Classification Evaluation for KNN ($k = 1, \ldots, 75$) on the 2d-Concentric-Circle Dataset. Optimal $k \approx 1 \rightarrow 10$. `tt_ratio=0.3`

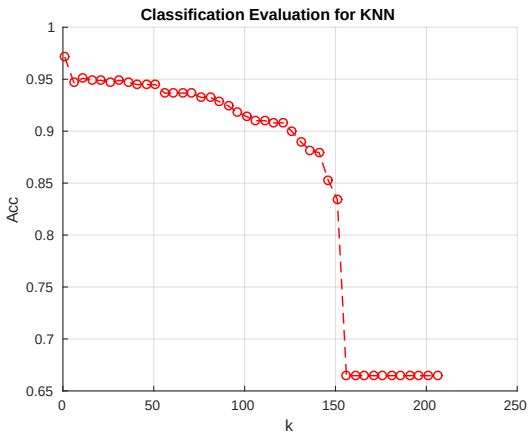

Figure 15: Classification Evaluation for KNN ($k = 1, \ldots, 210$) on the 9d-Breast-Cancer Dataset. Optimal $k \approx 1 \rightarrow 20$. `tt_ratio=0.3`

Further, the encrypted `test_knneval.p` function will compare the estimated accuracy curve from your `knn_eval.m` function with the `ML_toolbox` function. If you achieve the following, you can move on to part 2:

```
--- Testing my_accuracy.m ---
[Test] Accuracy implementation: Correct.
```

2 Part 2: Classification with K-Nearest Neighbors (KNN)

The Breast-Cancer-Wisconsin (Diagnostic) Dataset[2] is composed of $M = 698$ datapoints of $N = 9$ dimensions, each corresponding to cell nucleus features in the range of $[1, 10]$, with datapoints belonging to two classes $y \in \{\text{benign}, \text{malignant}\}$. In this part, we will apply KNN classification on this dataset and develop evaluation tools to assess the performances of the classifier.

2.1 K-NN for classification of high dimensional data

Load the Breast-Cancer-Wisconsin (Diagnostic) Dataset by running the **first** block of code in `test_knn_9D.m`. By running the **second** and **third** blocks you will split the datasets with your `split_data.m` and evaluate the classification accuracy with the function `knn_eval.m`.values that should generate a plot similar to Figure 15.

2.2 Confusion matrix

For real high-dimensional datasets, such as the Breast-Cancer-Wisconsin (Diagnostic) Dataset, estimating the classification accuracy is not a sufficient statistic to evaluate the classifiers performance. In these cases, one commonly computes a **confusion matrix** or error matrix, which is a specific table to visualize the performance in terms of classification of an algorithm. In this table, the rows represents the real classes of the data and the columns the estimated classes. The diagonal represents the well classified examples while the rest indicates confusions. In the case of a binary classifier (i.e. two classes such as the Breast-Cancer-Wisconsin Dataset), 4 values need to be computed to fill the confusion matrix. One of the labels is considered as positive and the other one as negative.

- **True positives (TP)**: number of test examples with a positive estimated label for which the actual label is also positive (good classification)
- **True negatives (TN)**: number of test examples with a negative estimated label for which the actual label is also negative (good classification)
- **False positives (FP)**: number of test examples with a positive estimated label for which the actual label is negative (classification errors)
- **False negatives (FN)**: number of test examples with a negative estimated label for which the actual label is positive (classification errors)

Table 1: Confusion matrix

		Estimated labels	
		Positive	**Negative**
Real labels	**Positive**	True positives (TP)	False negatives (FN)
	Negative	False positives (FP)	True negatives (TN)

[2]The Breast-Cancer-Wisconsin (Diagnostic) Dataset can be found in the UCI Machine Learning Repository: `https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+%28Diagnostic%29`

TASK 5: Implement confusion_matrix.m function (2pts)
Write a function that computes the confusion matrix of a binary classifier, given the real labels of the test dataset `y_test` and the labels estimated by the classifier `y_est`. NOTE: For our binary datasets, a positive label is when $\mathbf{y = 1}$ and a negative label is when $\mathbf{y = 0}$.

```
function [C] =  confusion_matrix(y_test, y_est)
%CONFUSION_MATRIX Implementation of confusion matrix
%   for classification results.
%   input ----------------------------------------------------------------
%
%       o y_test    : (1 x M), a vector with true labels y \in {0,1}
%                     corresponding to X_test.
%       o y_est     : (1 x M), a vector with estimated labels y \in {0,1}
%                     corresponding to X_test.
%
%   output ---------------------------------------------------------------
%       o C          : (2 x 2), 2x2 matrix of |TP & FN|
%                                             |FP & TN|.
```

Test Implementation

To test that the implementation if your confusion matrix is correct, run the **fourth** code block, the encrypted `test_confusionmatrix.p` function will compare the estimated confusion matrix from your `confusion_matrix.m` function with the `ML_toolbox` function. If you achieve the following, you can move on to the next task:

```
--- Testing confusion_matrix.m ---
[Test] Confusion Matrix implementation: Correct.
```

2.3 ROC curve

Based on the values of the confusion matrix, one can estimate a graphical representation of the classifier performance, the Receiver Operating Characteristic, so called ROC curve. The ROC curve plots the fraction of true positives and false positives over the total number of samples of class $\mathbf{y} \in \mathbf{1}, \mathbf{0}$ (positive,negative) in the dataset. Each point on the curve corresponds to a different value of the classifier's parameter (e.g. k). Figure 16 illustrates this concept.

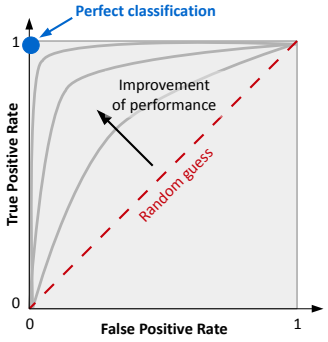

Figure 16: Illustration of the ROC curve.

To compute such a curve, one needs the **TPR** (True Positive Rates), otherwise known as *sensitivity* or *recall* and **FPR** (False Positive Rates), these are computed as follows [2]:

- **True Positive Rate (TPR)**:

$$TPR = \frac{TP}{TP + FN} = \frac{TP}{P} \tag{6}$$

- **False Positive Rate (FPR)**:

$$FPR = \frac{FP}{FP + TN} = \frac{FP}{N} \tag{7}$$

Where P are all positive values ($\mathbf{y} = \mathbf{1}$) and N are all negative values ($\mathbf{y} = \mathbf{0}$) [2].

TASK 6: Implement knn_ROC.m function (2pts)

This function shall compute the values for the ROC curve of K-NN for a range of k-values. This function must return two $(1 \times K)$ vectors, the first vector is the TPR of the classifier for each value of k and the second vector is the FPR. The columns correspond to the k-values.

```
function [ TP_rate, FP_rate ] = knn_ROC( X_train, y_train, X_test, y_test, ...
    k_range )
%KNN_ROC Implementation of ROC curve for kNN algorithm.
%   input ----------------------------------------------------------------
%       o X_train  : (N x M_train), a data set with M_test samples each being
%                    of dimension N each column corresponds to a datapoint
%       o y_train  : (1 x M_train), a vector with labels y \in {0,1}
%                    corresponding to X_train.
%       o X_test   : (N x M_test), a data set with M_test samples each being
%                    of dimension N each column corresponds to a datapoint
%       o y_test   : (1 x M_test), a vector with labels y \in {0,1}
%                    corresponding to X_test.
%       o k_range  : (1 x K), Range of k-values to evaluate
%
%   output ---------------------------------------------------------------

%       o TP_rate  : (1 x K), True Positive Rate computed for each value of k.
%       o FP_rate  : (1 x K), False Positive Rate computed for each value of k.
```

Test Implementation

The ROC curve for a binary classification problems is then defined as the two dimensional plot by representing **FPR** on its x-axis against **TPR** (sensitivity) on the y-axis. By running the **fifth** code block in `test_knn_9D.m` testing script you will be able to plot the resulting ROC curve. Rerun this code block a few times, using the same `tt_ratio = 0.1` to display the ROC curve for different randomizations of train and test datasets. You should obtain figures similar to Figure 17. To check that the implementation if your ROC curve is correct, the encrypted `test_knnROC.p` function will compare the estimated ROC curve for K-NN from your functions with the `ML_toolbox` function. If you achieve the following, you can move on to the next task:

```
--- Testing knn_ROC.m ---
[Test 1] TPR for KNN implementation: Correct.
[Test 2] FPR for KNN implementation: Correct.
```

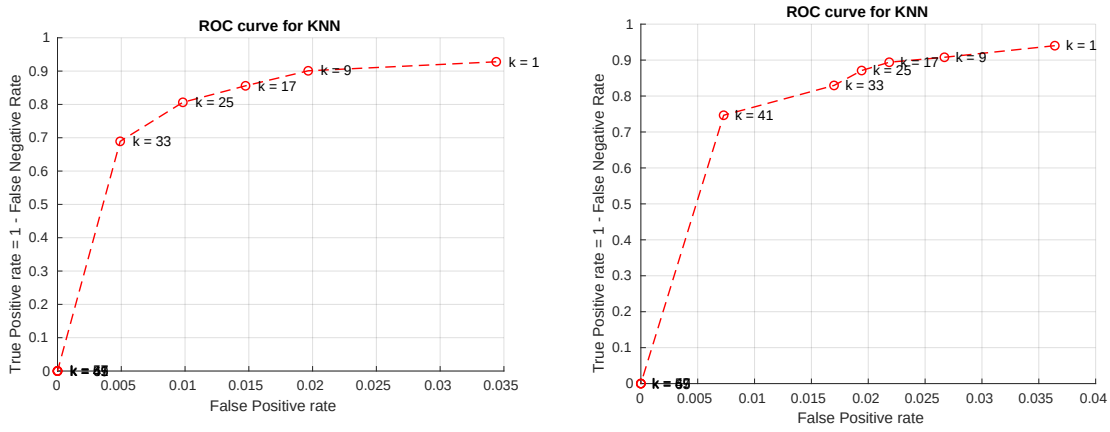

Figure 17: Two examples of ROC curves for K-NN generated with different randomization of train and test datasets for the same range of k-values. `tt_ratio=0.1` in the 2 cases.

2.4 Cross-validation

One can observe that the obtained ROC curve is very sensitive to the train dataset that is extracted by the function `split_dataset.m`. This observation is particularly true for kNN, because each example of the train set is directly used for classification. To assess the performances of the algorithm in a more robust way, a solution is to use cross-validation.
"Cross-validation refers to the practice of confirming an experimental finding by repeating the experiment using an independent assay technique" (Wikipedia). In Machine Learning, this consists of splitting the dataset several times at random into training and validation sets. The number of repetitions is referred to as the number of folds F. When one proceeds to 10 such random draws of training and testing sets, one says that one performs 10-fold cross-validation.

Here, we will use the function `split_data.m` to split the data F times at random between train and test while keeping the same `tt_ratio`. By averaging the resulting ROC curve across F, we will obtain a more robust assessment of the performance of our algorithm.

We will also compute the standard deviation σ across cross-validation runs using matlab function `std.m`. The standard deviation shows how the computed averages can vary (note that in normal distributions, 68.2% of the distribution takes place between $\pm\sigma$).

TASK 7: Implement cross_validation.m function (4pts)

```
1  function [TP_rate_F_fold, FP_rate_F_fold, std_TP_rate_F_fold, ...
       std_FP_rate_F_fold]  =  cross_validation(X, y, F_fold, tt_ratio, k_range)
2  %CROSS_VALIDATION Implementation of F-fold cross-validation for kNN algorithm.
3  %   input ----------------------------------------------------------------
4  %
5  %       o X          : (N x M), a data set with M samples each being of
6  %                           dimension N each column corresponds to a datapoint
7  %       o y          : (1 x M), a vector with labels y \in {0,1}
8  %                           corresponding to X.
9  %       o F_fold     : (int), the number of folds of cross-validation to compute.
10 %       o tt_ratio   : (double), Training/Testing Ratio.
11 %       o k_range    : (1 x K), Range of k-values to evaluate
12 %
13 %   output ---------------------------------------------------------------
14 %
15 %       o TP_rate_F_fold  : (1 x K), True Positive Rate computed for each
16 %                           value of k averaged over the number of folds.
17 %       o FP_rate_F_fold  : (1 x K), False Positive Rate computed for each
```

```
%                             value of k averaged over the number of folds.
%       o std_TP_rate_F_fold  : (1 x K), Standard Deviation of True Positive
%                               Rate computed for each value of k.
%       o std_FP_rate_F_fold  : (1 x K), Standard Deviation of False Positive
%                               Rate computed for each value of k.
%
```

Test Implementation

Run the **last** code block in `test_knn_9D.m` to plot the resulting ROC curve created from F-fold cross-validation. You should obtain a curve similar to the Fig.18, which is a plot of the ROC curve displaying standard deviations for each k across cross-validation runs.

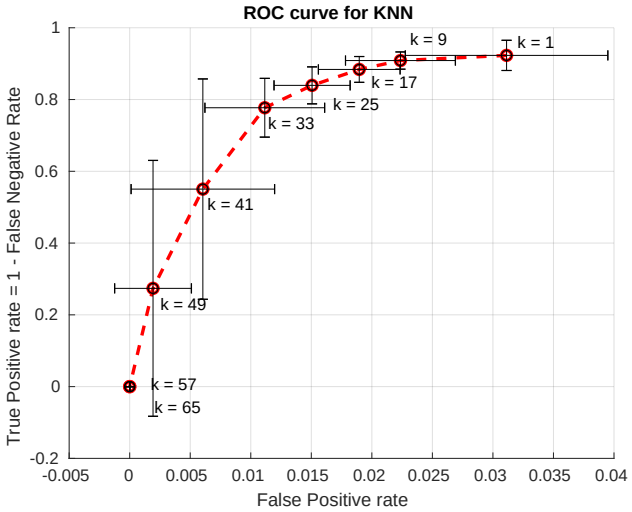

Figure 18: ROC curve with standard deviations for kNN with 10-fold cross-validation. `tt_ratio=0.1`

To check that the implementation if your ROC curve through cross-validation is correct, the encrypted `test_cross_validation.p` function will compare your functions with the `ML_toolbox` functions. If you achieve the following, you can move on to the next task:

```
--- Testing cross_validation.m ---
[Test] Cross Validation implementation: Correct.
```

References

[1] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification (2nd Edition)*. Wiley-Interscience, 2000.

[2] Matjaz Majnik and Zoran Bosnic. Roc analysis of classifiers in machine learning: A survey. *Intell. Data Anal.*, 17(3):531–558, 2013.