Machine Learning Programming

Assignment 3: K-Nearest Neighbors (KNN)

This series of practicals focus on the development of machine learning algorithms introduced in the Applied Machine Learning course. In this practical, you will code the K-Nearest Neighbors (KNN) algorithm in Matlab and its applications.

Deadline: November 15, 2022 @ 8pm

Assignments must be turned in by the deadline. **1/6 point will be removed for each day late (a day late starts one hour after the deadline).**

Procedure: From the course Moodle webpage, you must download and extract the `.zip` file named `TP3-KNN-Assignment.zip`. The folder contains the following elements:

- tp3_knn_part1.m
- tp3_knn_part2.m
- **functions**
  - **part1**
  - **part2**
- **plot_functions**
- **data**
- **evaluation_functions**
- **utils**

Only the elements in blue need to be filled out. The rest are helper functions or data to be imported. The folder **evaluation_functions** contains encrypted functions that evaluate the code written and provide feedback on the results. In this assignment, only the evaluation functions for the `knn` function in part 1, i.e. the main algorithm, are provided. For the other functions you can compare your results with the graphs.

You must submit an archive as a `.zip` file with the following naming convention `TP3-KNN-SCIPER.zip`, where `SCIPER` need to be replaced by your own **sciper number**. You must submit **only** the files/folders in blue. **DO NOT INCLUDE** the folders **plot_functions**, **data**, **utils** and **evaluation_functions**.

We take plagiarism very seriously. Submitting works that are not yours will result in a report submitted to the competent EPFL authorities.

Part 1: K-Nearest Neighbors (KNN)

KNN (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.

KNN 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.). This makes it applicable to a lot of dataset. It is 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).

KNN relies on a similar mechanism as DBSCAN, presented in the Applied Machine Learning course, but 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. KNN does not have these two additional mechanisms and is hence very sensitive to outliers.

Data Handling for Classification

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 validation ratio: `valid_ratio`, where `valid_ratio=0.3` corresponds to 30% of your points used for **testing** (or validation) and the rest of your points used for **training**. One can vary the value to have a better estimate of the classifier's performance. In general, validation ratios are tested in the range of $\{30, 40, 50, 60, 70\}$, depending on how much data you have. Special considerations of train/test sets:

- A large validation or test set generally gives a more reliable estimate of the classifier's accuracy (i.e. a lower variance estimate).
- A large training set will generally be more representative of how much data we actually have for learning process.
- Using only a training set (no validation set) does not give a reliable accuracy estimate. It cannot tell how sensitive the classifier is wrt. specific samples.

The splitting function has already been implemented during the exercise sessions and, therefore, is given in the **utils** folder.

KNN Algorithm

In classification problems we are given a training dataset $D = \{(\mathbf{x}^1, y^1), (\mathbf{x}^2, y^2), \dots, (\mathbf{x}^M, y^M)\}$ where $\mathbf{x}^i \in \mathbb{R}^N$ is the i-th N-dimensional data point (or feature vector) for $i = 1, \dots, M$ and $y^i \in \mathbb{N}$ is the categorical outcome (or class label) corresponding to each data point. Generally, the labels are a sequence of integer ranging from 1 to C, where C is the number of classes (you might also find some datasets labelled from 0 to $C-1$). The goal is then, given a new sample (or query point) $\mathbf{x}' \in \mathbb{R}^N$ we would like to predict its label $\hat{y}' \in \mathbb{N}$\footnote{We use $\hat{}$ for $\hat{y}$ to indicate that it is an estimated value and not the true label y.}. 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.:

$\mathbf{x}'$ 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:

- Calculate the pairwise distances between $\mathbf{x}'$ and all points in the training dataset $D_x = \{\mathbf{x}^1, \dots, \mathbf{x}^M\}$
- Extract the k nearest neighbors of $\mathbf{x}'$ from the pairwise distances computed in step 1 $\mathbf{x}'_k = \{x_1, \dots, x_k\}$ and their corresponding class labels $\mathbf{y}^k = [y^1, \dots, y^k]$
- 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 $\mathbf{x}'$ 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.

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.

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 $D_x = \{\mathbf{x}^1, \dots, \mathbf{x}^M\}$.

Such a distance function has already been implemented in the `compute_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 mostly use the Euclidean distance:

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

which should be called as `compute_distance(x1, x2, 'L2')`. In Part 2 of the assignment you will also implement another metric for dataset that combines continuous and categorical data. For ease of use, we provide the correct implementation of this function in the **utils** folder.

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,

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

where $\mathbf{d} \in \mathbb{R}^M$ is the result of the distance function already implemented and applied on each points of the training dataset.

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

$$\mathbf{d}_{(s)} = \{d^i_{(s)} \quad | \quad d^i_{(s)} < d^{i+1}_{(s)} < \dots < d^M_{(s)}\} \qquad (3)$$

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

$$y_{knn} = \{y^{I(d^1_{(s)})}, y^{I(d^2_{(s)})}, \dots, y^{I(d^k_{(s)})}\}, \qquad (4)$$

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

Majority Vote

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

$$\hat{y}' = argmax(\{\sum y_{knn} = c_1, \sum y_{knn} = c_2, \dots \sum y_{knn} = c_N\}), \qquad (5)$$

where $\{c_1, c_2, \dots c_N\}$ are the labels associated to each classes C. Now we will implement these three steps in the `knn.m` function.

Task 1: Implement the `knn.m` function (6pts):

This function has the following signature:

In [ ]:
```
function [ y_est ] =  knn(X_train,  y_train, X_test, params)
```

where `X_train` and `y_train` are the features and labels on which to train your model respectively and `X_test` is a testing dataset for which you estimate the labels and return them in `y_est`. The variable `params` is a **structure array** that contains the hyper-parameters of the algorithm such as the type of the distance function `d_type` or the number of neighbors `k`. Accessing a field of this structure array is possible via `params.the_field`, e.g. `params.d_type` to get the distance type of `params.k` to get the number of neighbors.

Implementation Hint: Useful functions: `compute_distance`, `sort` and `unique`.

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,

$$acc = \frac{\sum_{i=1}^{M_{test}} \delta_{(y'_i, \hat{y}'_i)}}{M_{test}} \qquad (6)$$

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

Task 2: Implement `accuracy.m` function (1pt)

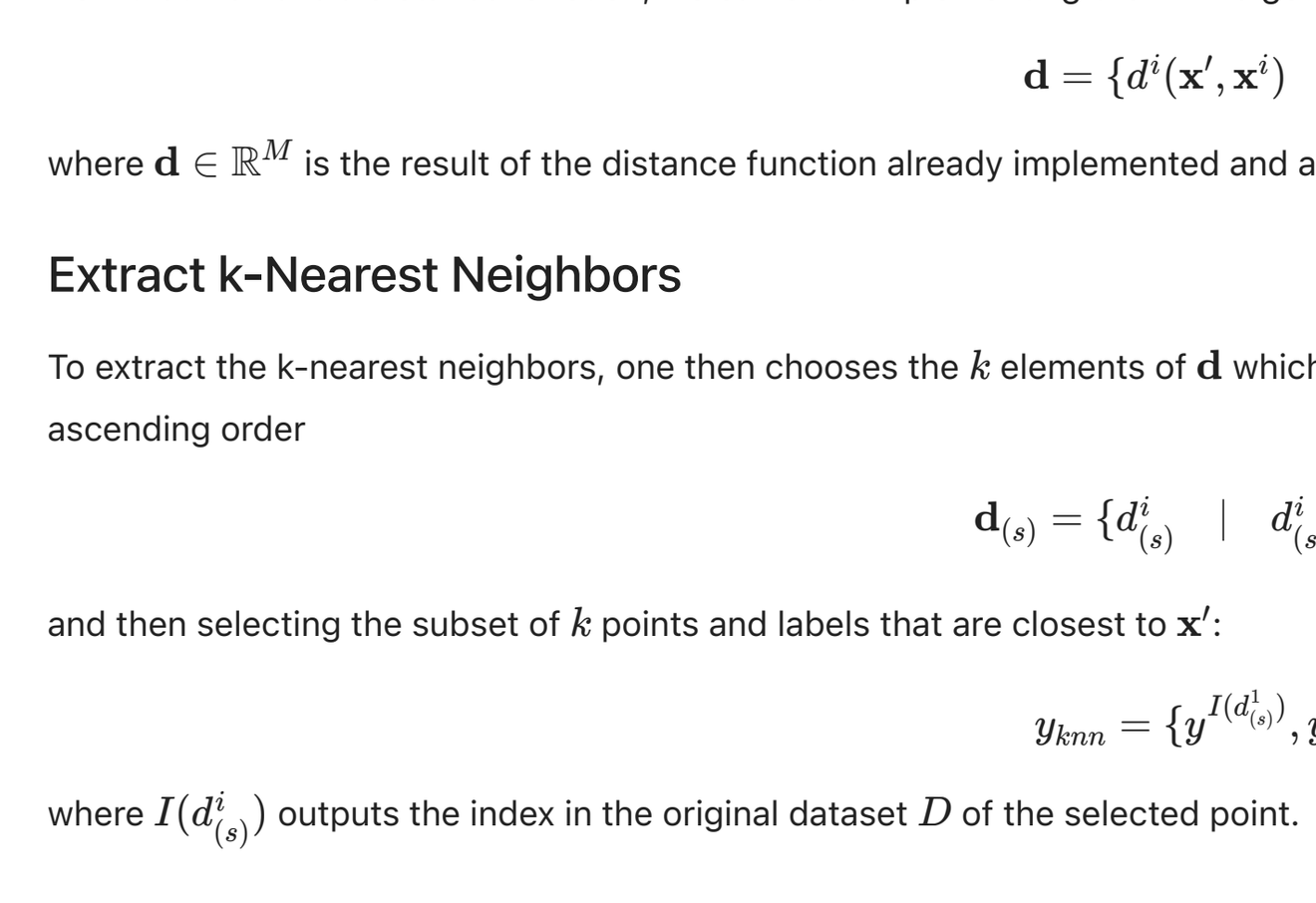
This function has the following signature:

In [ ]:
```
function [acc] =  accuracy(y_test, y_est)
```

Visualizing the results

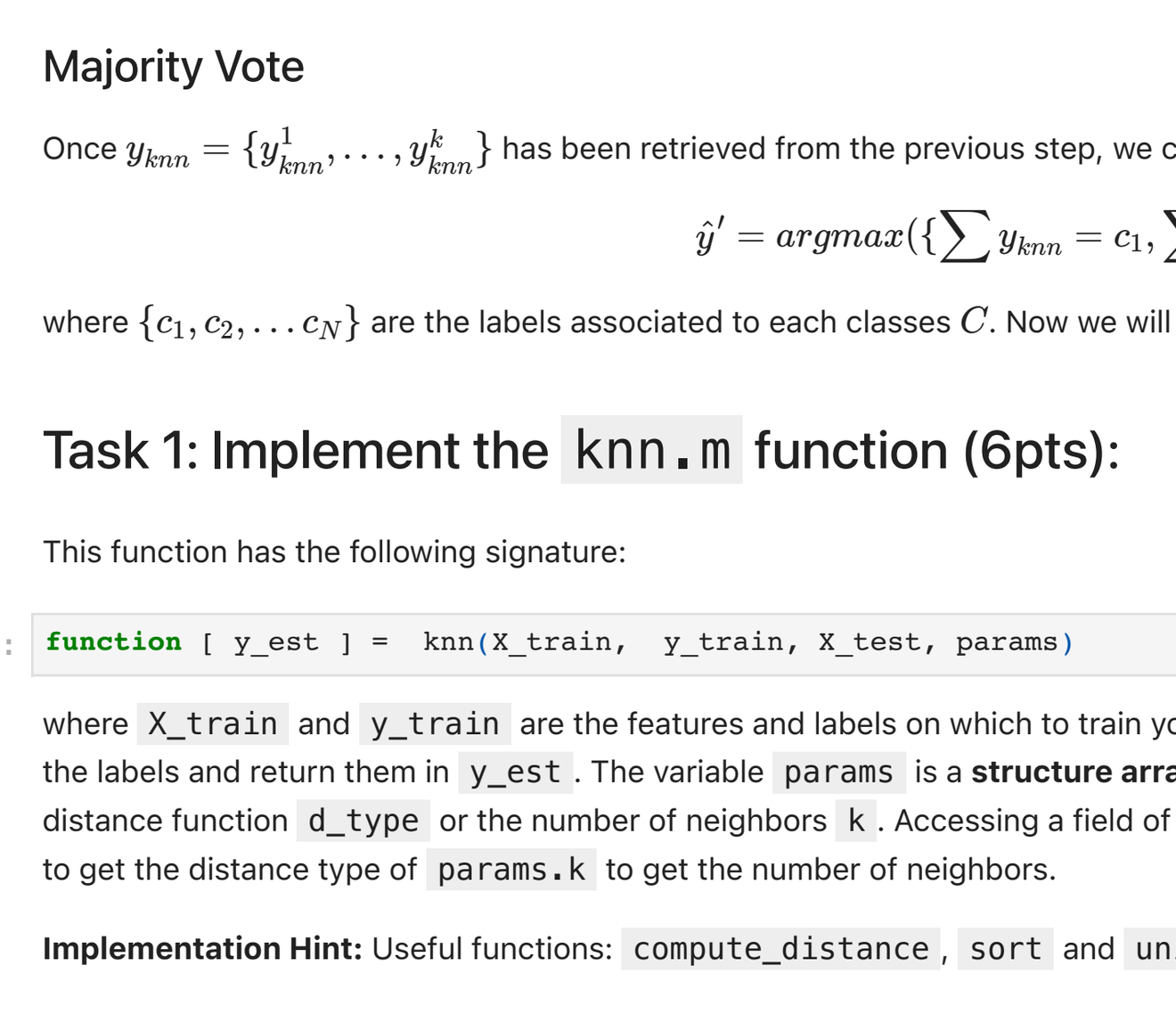
By running the `tp3_knn_part1.m` code block, you will be able to visualize the results of `knn` and `accuracy` functions, e.g. for $k = 1$:

In [8]:
```
plot_dataset(X, y, X_test, y_est, valid_ratio, params);
```



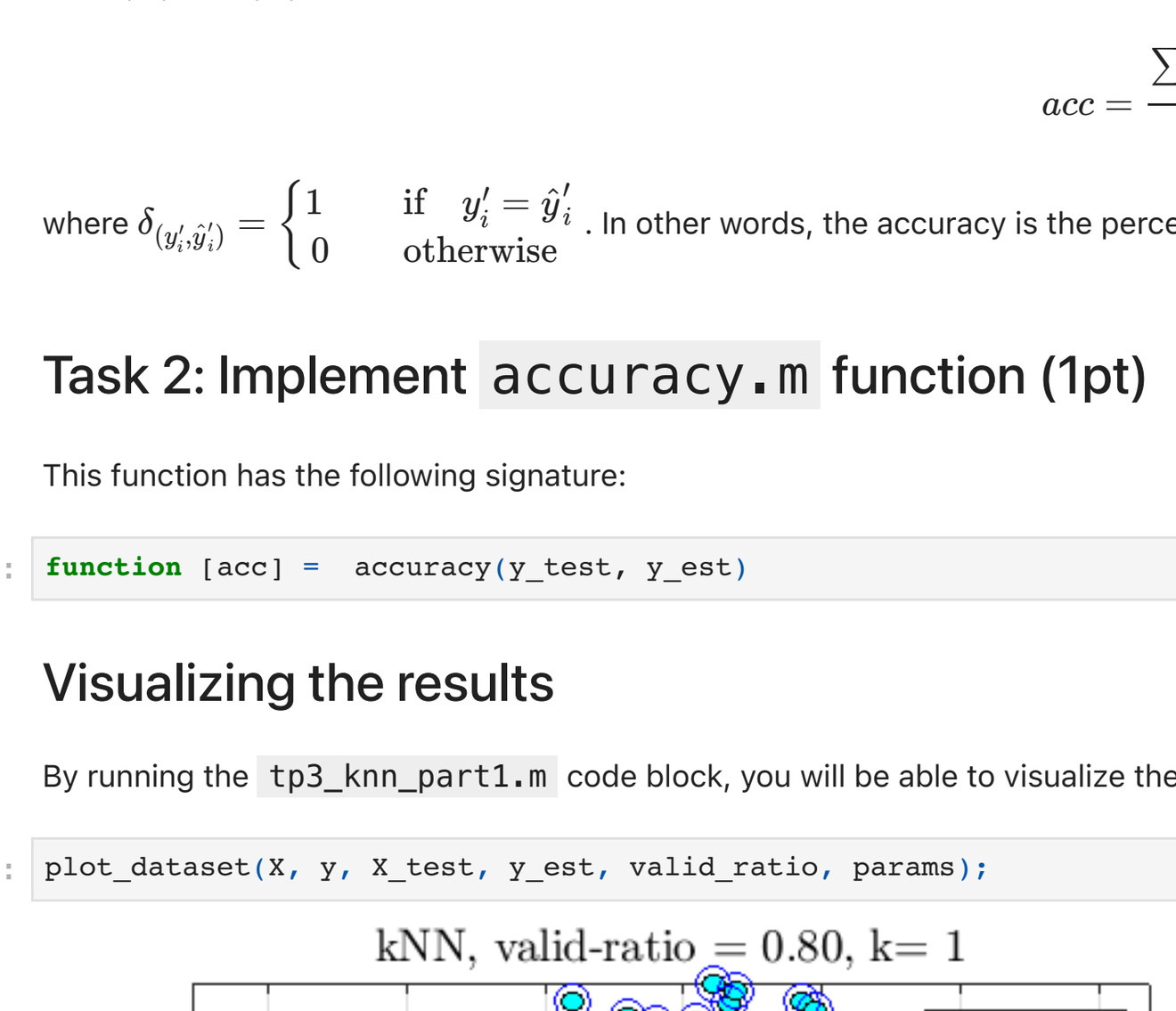
You can modify k to have the following values $k = \{1, 5, 15, 35, 61, 75\}$, to see how KNN behaves on the same dataset. You must take into consideration that due to the random data split you can have different values of `accuracy` for the same K and plots that look different. By running the decision-boundary code block, one can visualize the decision boundaries for each output:

In [17]:
```
params.k = 1;
plot_boundaries(X, y, X_train, y_train, valid_ratio, params)
```

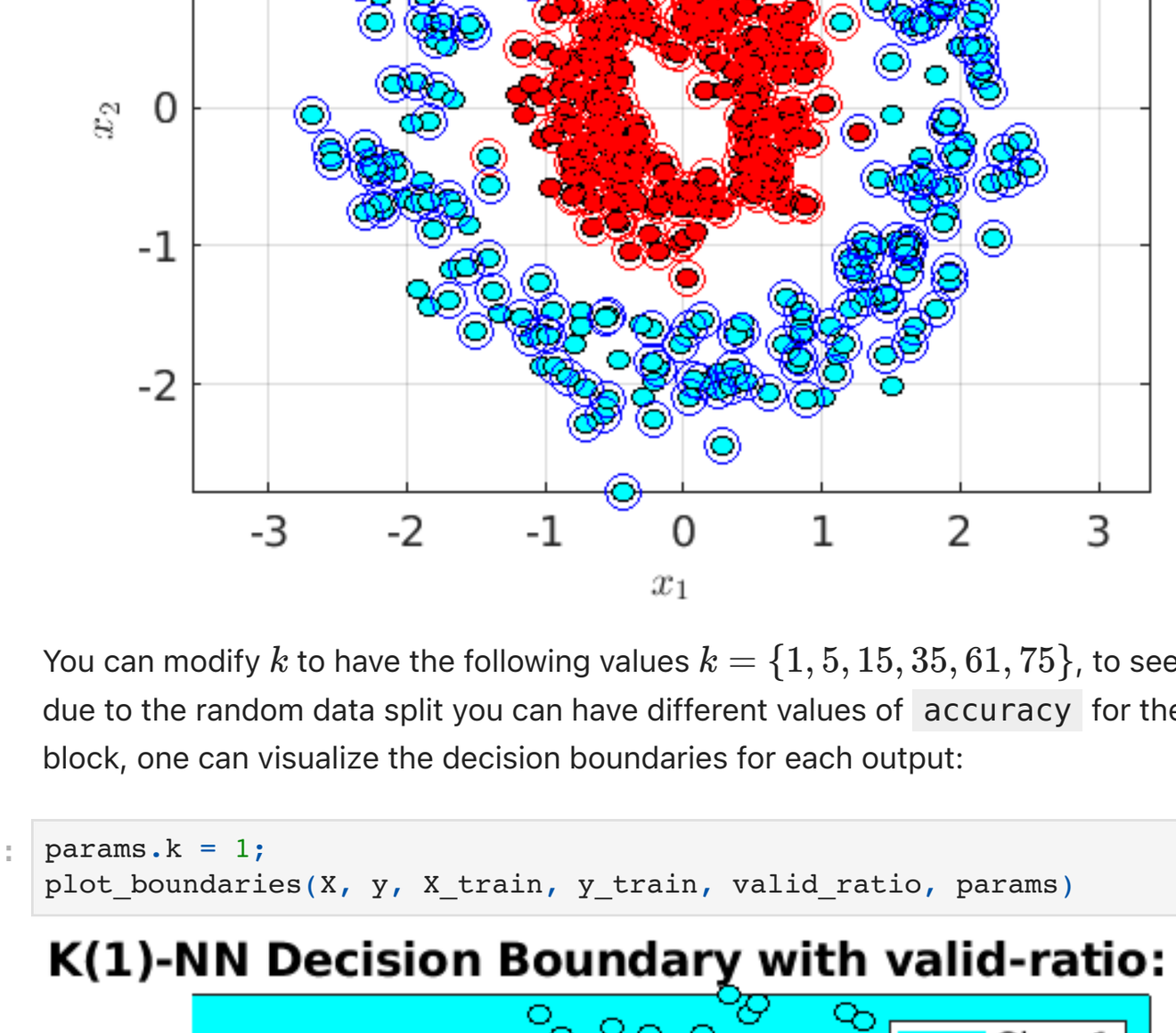


NOTE: Given the random nature of the `split_data` function, the following plots don't necessarily have to be the same. In fact, for a high `valid_ratio`, i.e. `valid_ratio` > 0.7 and a high K, i.e. $K > 30$ this will undoubtedly be the case.

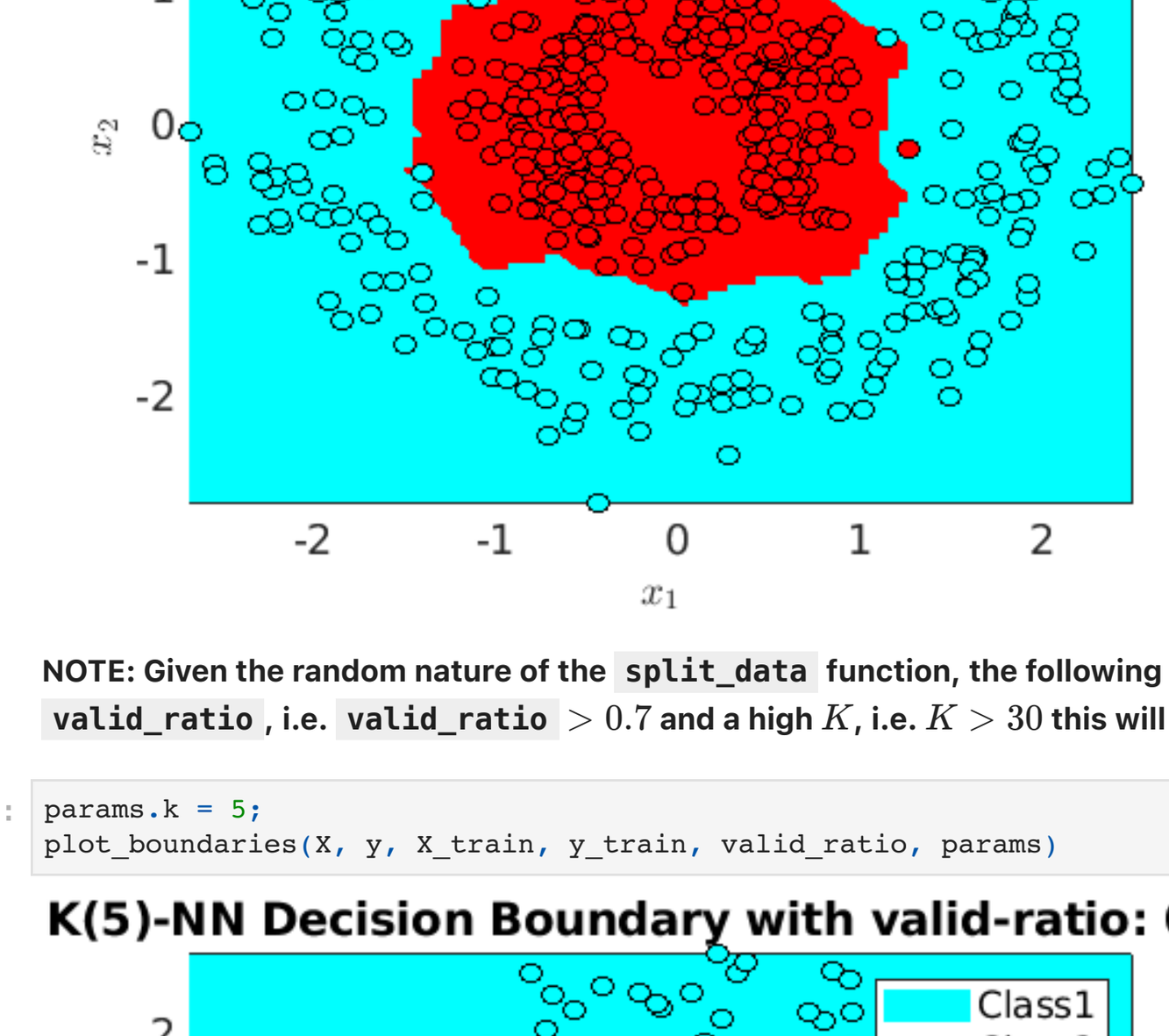
In [19]:
```
params.k = 5;
plot_boundaries(X, y, X_train, y_train, valid_ratio, params)
```



In [21]:
```
params.k = 20;
plot_boundaries(X, y, X_train, y_train, valid_ratio, params)
```



In [23]:
```
params.k = 60;
plot_boundaries(X, y, X_train, y_train, valid_ratio, params)
```



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.

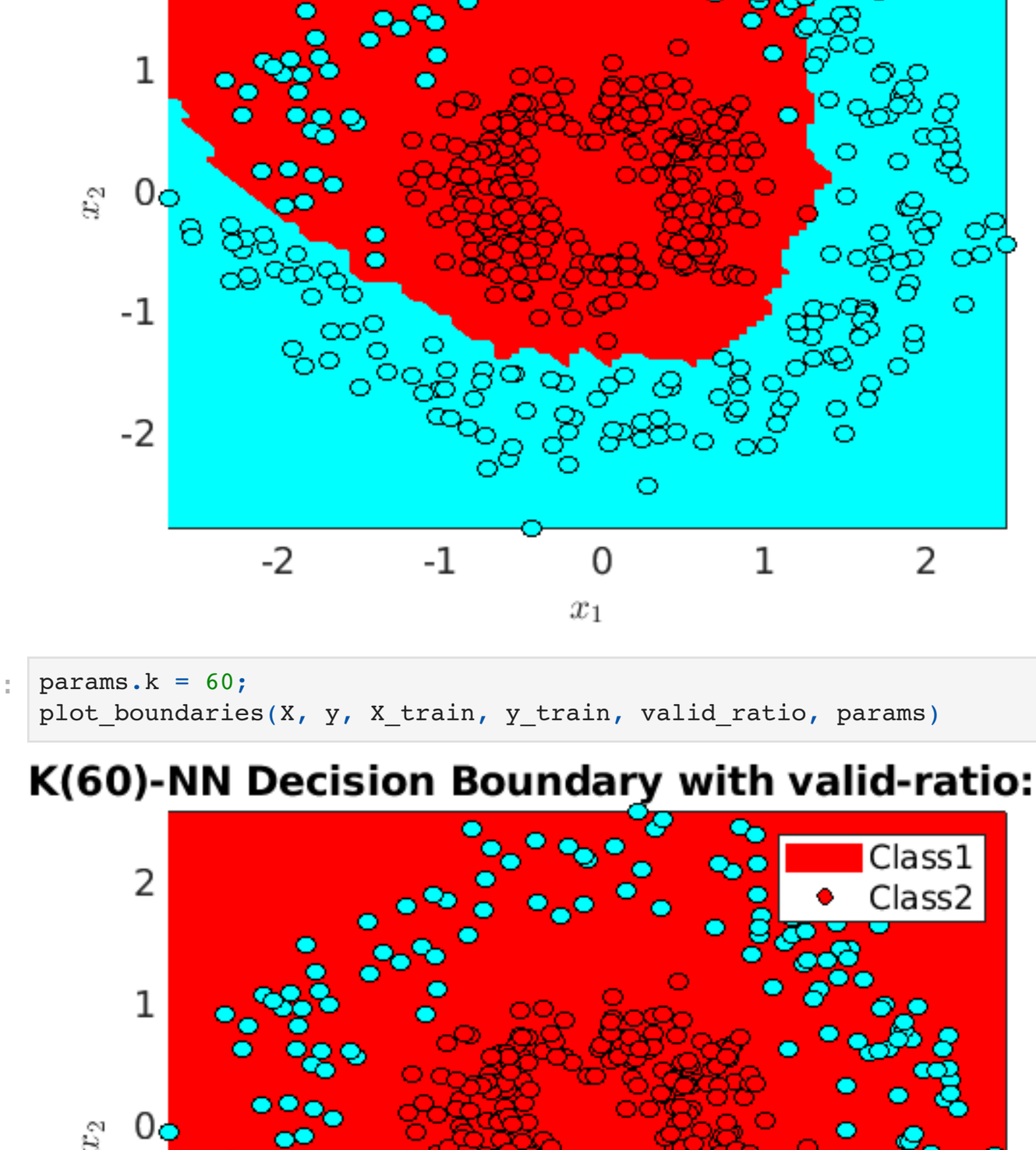
Task 3: Implement `knn_eval.m` function (2pts)

This function has the following signature:

In [ ]:
```
function [acc_curve] = knn_eval( X_train, y_train, X_test, y_test, k_range, params)
```

The script `tp3_knn_part1.m` includes a visualization of the accuracy curve for different values of k in `k_range`

In [29]:
```
plot_accuracy(params.k_range,acc_curve)
```



As you can see in the plot, increasing the number of neighbors might simply degradate the performances.

Part 2: Classification with KNN

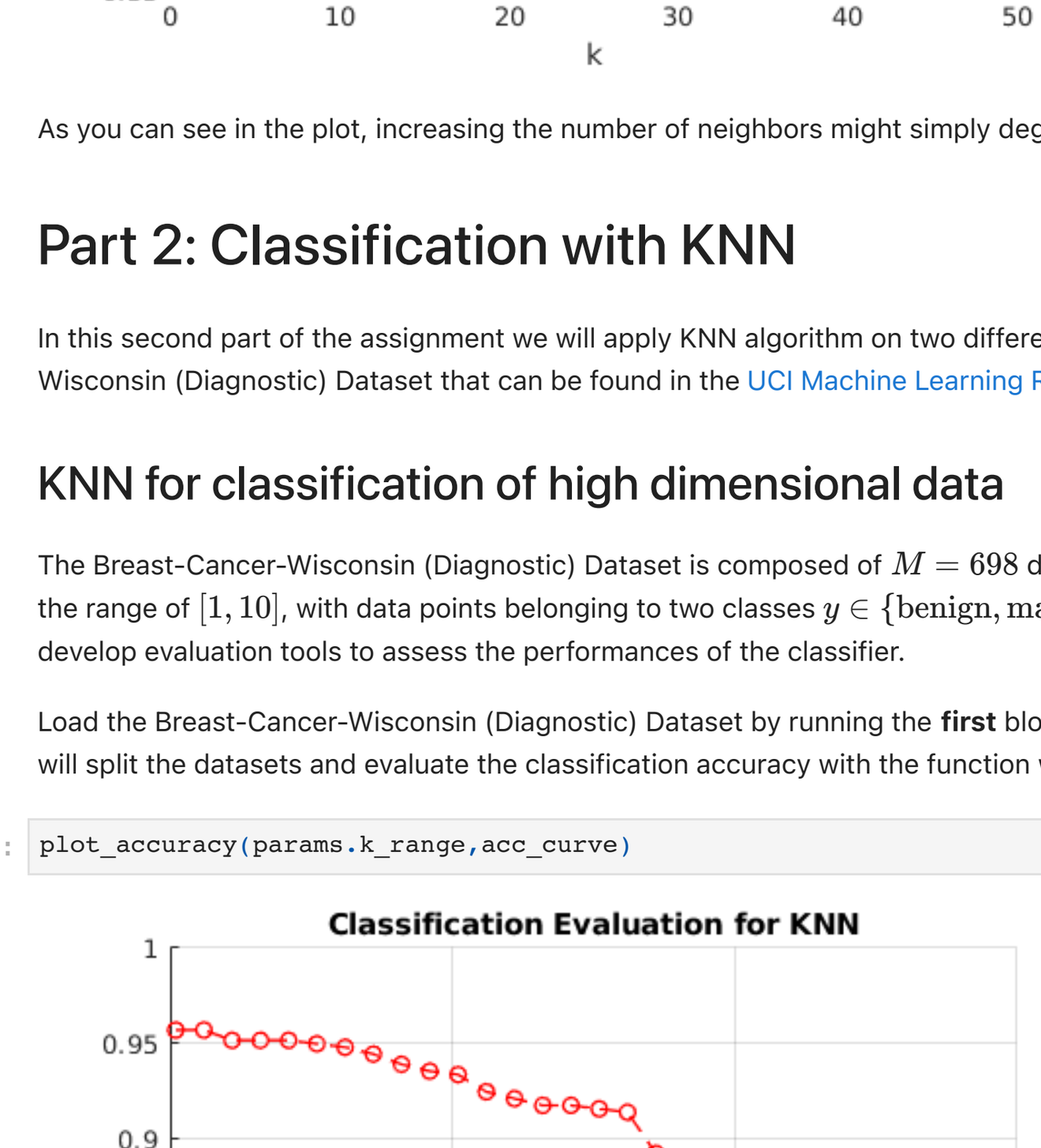
In this second part of the assignment we will apply KNN algorithm on two different datasets. One with high-dimensional continuous data, the Breast-Cancer-Wisconsin (Diagnostic) Dataset that can be found in the UCI Machine Learning Repository

KNN for classification of high dimensional data

The Breast-Cancer-Wisconsin (Diagnostic) Dataset is composed of $M = 698$ data points of $N = 9$ dimensions, each corresponding to cell nucleus features in the range of $[1, 10]$, with data points 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.

Load the Breast-Cancer-Wisconsin (Diagnostic) Dataset by running the **first** block of code in `tp3_knn_part2.m`. By running the **second** and **third** blocks you will split the datasets and evaluate the classification accuracy with the function written previously.

In [6]:
```
plot_accuracy(params.k_range,acc_curve)
```



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)

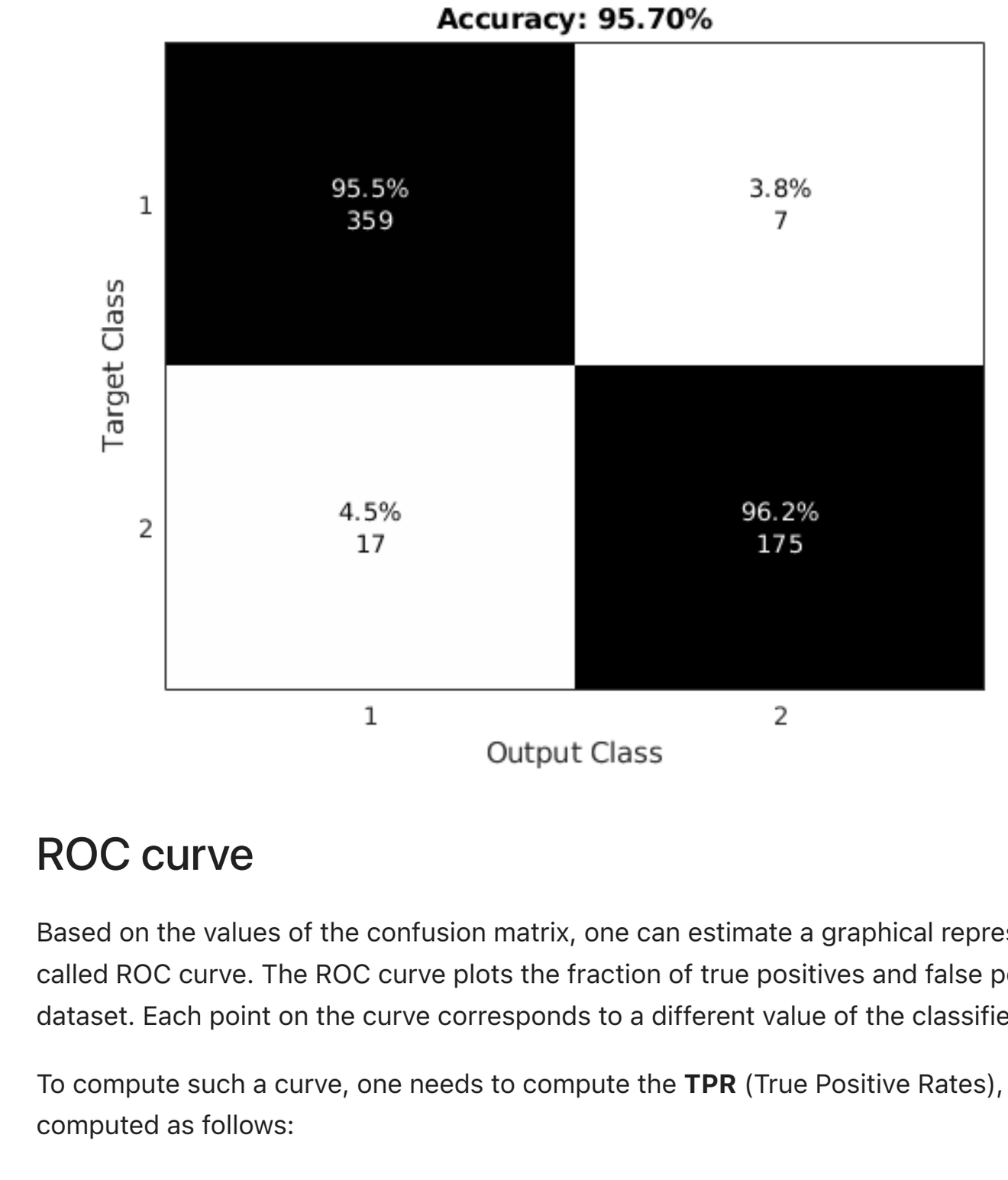
Task 4: Implement `confusion_matrix.m` function (2pts)

This function has the following signature:

In [ ]:
```
function [C] = confusion_matrix(y_test, y_est)
```

Plotting this confusion matrix provides the following results, where the target class is the true class, and the output class is the estimated class:

In [6]:
```
plotConfMat(C);
```



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 $\mathbf{y}$ (positive,negative) in the dataset. Each point on the curve corresponds to a different value of the classifier's parameter (e.g. k).

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

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

$$TPR = \frac{TP}{TP + FN} = \frac{TP}{P} \qquad (7)$$

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

$$FPR = \frac{FP}{FP + TN} = \frac{FP}{N} \qquad (8)$$

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

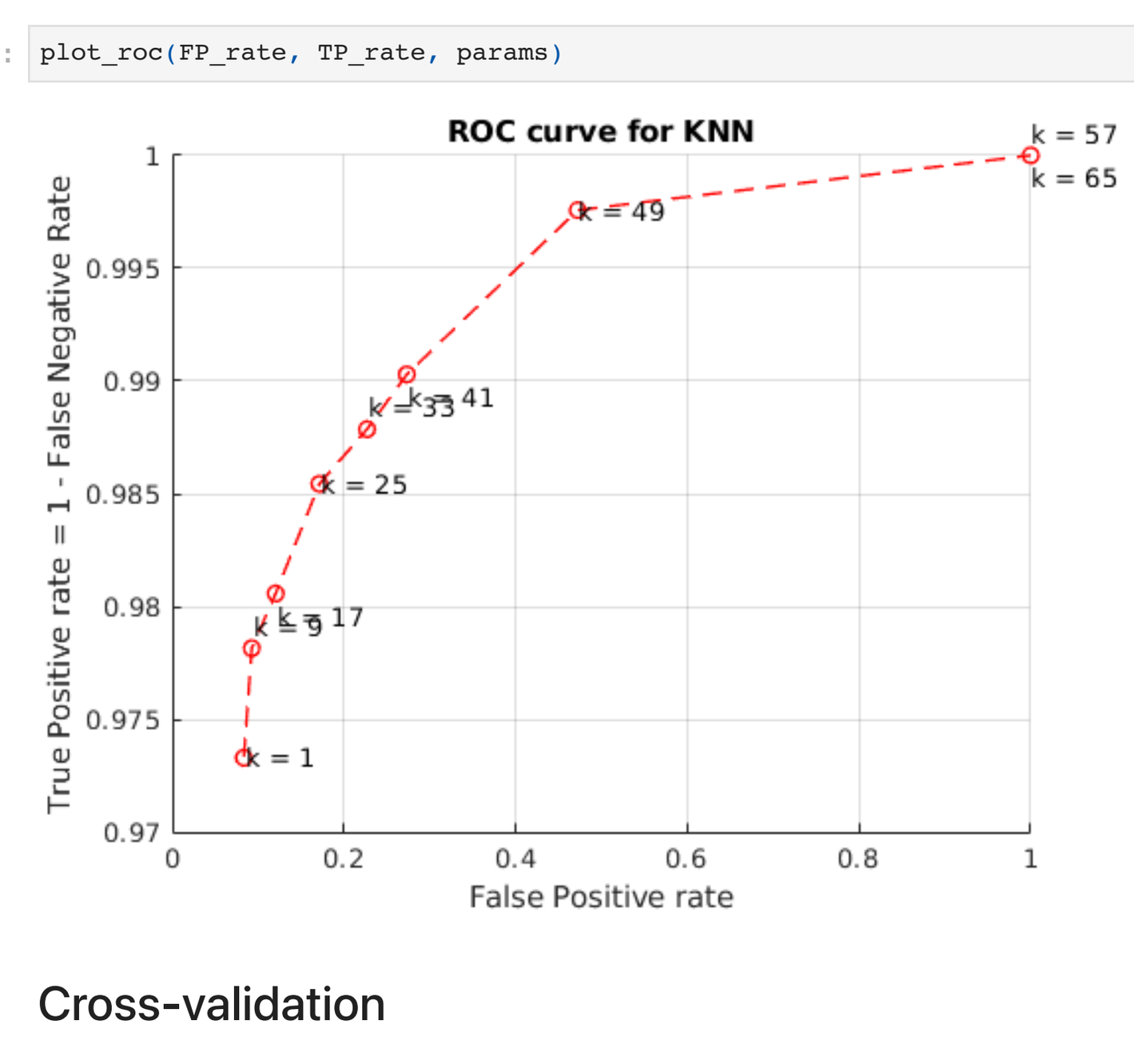
Task 5: Implement `knn_ROC.m` function (2pts)

This function has the following signature:

In [ ]:
```
function [ TP_rate, FP_rate ] = knn_ROC( X_train, y_train, X_test, y_test,  params )
```

It shall compute the values for the ROC curve of K-NN for a range of k-values. It 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.

In [8]:
```
plot_roc(FP_rate, TP_rate, params)
```



Cross-validation

One can observe that the obtained ROC curve is very sensitive to the train dataset that is extracted by the function `split_data`. 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` to split the data F times at random between train and test while keeping the same `valid_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. The standard deviation shows how the computed averages can vary (note that in normal distributions, 68.2% of the distribution takes place between $\pm\sigma$).

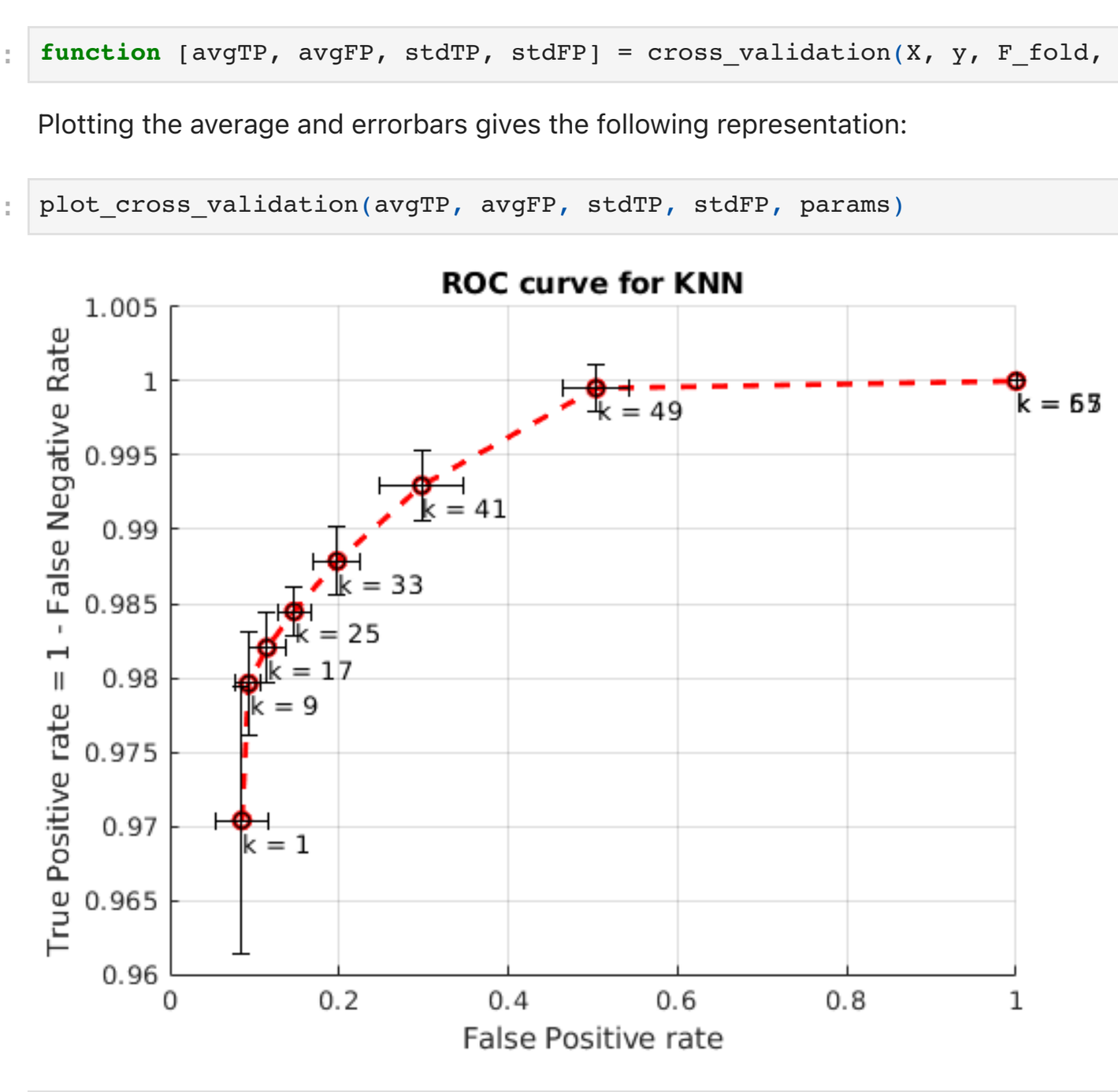
Task 6: Implement `cross_validation.m` function (4pts)

This function has the following signature:

In [ ]:
```
function [avgTP, avgFP, stdTP, stdFP] = cross_validation(X, y, F_fold, valid_ratio, params)
```

Plotting the average and errorbars gives the following representation:

In [14]:
```
plot_cross_validation(avgTP, avgFP, stdTP, stdFP, params)
```



In [ ]: