Report for the Machine Learning course

Ramona Ferrari
Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi
DIBRIS
Genoa, Italy
s4944096@studenti.unige.it

***Abstract*—This document is a report for the Machine Learning course. Each assignment treats different topics: Naive Bayes Classifier, Linear Regression, KNN Classifier and Neural Networks. After the background statement and also the problem statement, the author discusses details about the methods used and description of the assignments, followed by the results and the conclusions.**

I. INTRODUCTION ABOUT NAIVE BAYES CLASSIFIER

A. Bayes's theorem

The Naive Bayes Classifier is a supervised machine learning algorithm, which is used for classification. It is supervised, because it contains both input and target. The classifier is called "Naive", because it assumes the input variables are all independent of each other; mathematically:

$$Pr(x_1, ..., x_d|t_i) = Pr(x_1|t_i)Pr(x_2|t_i)...Pr(x_d|t_i) \quad (1)$$

where:

- x_i is the observation;
- t_i is the target;

The classifier is based on an important theorem called **Bayes's theorem**, which states:

$$P(t_i|x) = \frac{P(x|t_i)P(t_i)}{P(x)} = \frac{P(x|t_i)P(t_i)}{\sum_{j=0}^{N} P(x|t_j)P(t_j)} \quad (2)$$

where:

- x is an experimental observation;
- t_i is a set of alternative hypothesis;
- $P(x)$, called marginal probability of x, is the probability of observing x in any case;
- $P(t|x)$ is the "a posteriori probability";
- $P(x|t)$ is the likelihood of observing x when t holds.

The naive assumption may not always hold true in real-world scenarios. However, this assumption allows the algorithm to be computationally efficient and makes it suitable for large data sets. The naive assumption makes the (2) in this way:

$$P(t_i|x) = P(t_i) \prod_{j=0}^{N} P(x_j|t_i) \quad (3)$$

It learns the statistical relationships between the features and the classes from the data, and then uses this knowledge to make predictions on unseen data.

B. Laplace smoothing

Laplace smoothing, also known as *add-one smoothing*, is a technique used in natural language processing and machine learning to solve a problem of missing data. In essence, Laplace smoothing involves adding a small constant value to the count of each possible outcome, both observed and unseen, when estimating probabilities. By doing this, it ensures that no probability estimate is zero and that even unseen events have a non-zero probability assigned to them.
Mathematically, Laplace smoothing can be represented as:

$$P(x = i) = \frac{n_i + a}{N + av} \quad (4)$$

where:

- n_i is the number of occurrences of n_i in the data;
- a is the parameter of Laplace smoothing;
- N is the number of observation;
- v is total number of possible outcomes.

The additive constant a has an important meaning with respect to its value. Larger values of a result in more smoothing, while smaller values of a lead to less smoothing. The choice of the value of a depends on the specific application and data characteristics. Smaller values (precisely $a < 1$) might be more appropriate when the training data is large and representative, while larger values ($a > 1$) could be suitable when the training data is limited or imbalanced.

C. ASSIGNMENT 1

The tasks are:

1) Data Preprocessing
2) Build a Naive Bayes Classifier
3) Improve the classifier with Laplace (additive) smoothing

D. Data Preprocessing

The data set is about the weather and is given in the following structure (in order of columns) and with the listed possible levels:

- Outlook: overcast, rainy, sunny;
- Temperature: hot, cool, mild;
- Humidity: high, normal;

- Windy: true, false;
- Play: yes, no.

The target ”play” determines whether the weather conditions permit outdoor play. Since the software used is Matlab, it is advantageous to convert all levels into integers greater than 1. Matlab is very efficient in manipulating matrices and vectors. To convert the categorical values of the columns, sequential numbering can be employed. However, it is also acceptable to repeat numbers within columns, and this does not result in any loss of data. Subsequently, a script needs to be written where the data is arranged in a matrix, and then this matrix is divided into two separate matrices for training and testing purposes. It’s essential to verify that both matrices have the same dimensions in terms of columns, and the data values are greater than one to avoid errors. In this specific scenario, the training matrix is of size $10\,\mathrm{x}\,5$, while the testing matrix is of size $4\,\mathrm{x}\,5$.

E. Build a Naive Bayes Classifier

To build a Naive Bayes Classifier it’s fundamental to adapt the (3) to the data and understand which terms are in the considered data set:

- t_i are the possible values of the target ”play”, in this case play outside or not;
- x are the set of values assumed by the remaining columns.

Firstly, all the required probabilities are computed on the training matrix, and the outcomes are placed into two distinct matrices, corresponding to the two possible values of the target play (conditional probabilities). Subsequently, these values can be utilized to compute $P(t_i|x)$ on the test matrix and compare the obtained results with the actual values of the target play. This final step is referred to as *Classification*.
To calculate the prior probability of ”yes” and ”no,” it is essential to count the frequency of occurrences of ”yes” (or ”no”) in the matrix and divide the value by the total number of rows in said matrix.

```
% Computation of P(ti) and P(tj) on A

prior_yes = count_1 / num_rows_A; % priori probability of "yes"
prior_no = count_2 / num_rows_A; % priori probability of "no"
```

Fig. 1: Computation of priori probabilities

F. Improve the classifier with Laplace (additive) smoothing

To solve a problem of missing data, you can modify the computation of conditional probabilities adding the value of v and the constant a (choosing an appropriate value). So the code would be:

This code effectively counts the number of occurrences of a specific value in a column and checks if the target play is equal to 2, representing ”no” for the conversion in this case. The count is stored in a variable called *countno*, which is then used to implement the (4). The resulting numbers are stored in a matrix called $prob\_matrix\_no$.
The function *Numel* in Matlab is utilized to determine the

```
for i = 1:width(A)-1
    a = unique(A(:,i));
    countno = zeros(size(a));
    for j=1:height(A)
        for z = 1:size(a)
            if (A(j,i) == a(z) && A(j, width(A)) == 2)
                countno(z) = countno(z) + 1;
            end
        end
    end

countno = (countno + alpha) / (count_2 + alpha*numel(a));

prob_matrix_no(1:size(countno),i)= countno;
end
```

Fig. 2: Code with Laplace Smoothing

number of elements in the array a, which has already been defined as the unique values in the test matrix, denoted as A. To avoid any confusion with this existing variable, the additive constant of the Laplace smoothing equation is named $alpha$. This code segment focuses on computing the conditional probabilities associated with the outcome ”no”, but the same code is employed for calculating probabilities related to the outcome ”yes”.

II. Conclusion about Naive Bayes Classifier

The concept or **Error Rate** is introduced to better understand how the classifier performs. It works on comparing the result of the classification and the true values of the target play. The error rate can be:

- 0% means the classifier has predicted all entries correctly;
- 25% means the classifier predicted 3 entries correctly;
- 50% means the classifier predicted half entries correctly;
- 75% means the classifier predicted 1 entry correctly;
- 100% means the classifier has predicted all entries wrongly.

In this case, the classifier has an average error rate of 30% over 10 attempts.
To make the code more readable, you can define a function to compute the matrices which represent the conditional probabilities, instead of rewriting the same part of the code. It’s possible also to write a *for loop* to run the script more than one time, collect the results and estimate how good it is the classifier on a certain number of attempts, instead of doing it manually. The main point of this assignment is to identify the terms involved in the (3) and compute them in the easily possible way.

A. Think further

If you have large data sets, you may incur in numerical errors while multiplying a lot of frequencies together. A solution is to work with log probabilities, by transforming logarithmically all probabilities and turning all multiplications into sums.
To proceed if the input variables were continuous, we would

need to use *probability density functions* (PDF) instead of probability mass functions (PMF).
To compute probabilities based on continuous variables, we would need to estimate the parameters of the PDFs from the data. Once the PDFs are estimated, we can use them directly to compute probabilities for new observations.
In the context of the Naive Bayes Classifier, we would need to modify the assumption of independence among features. With continuous variables, it is unlikely that they would be completely independent, so we would need to consider the correlations among the variables when computing the likelihoods in Equation (3). This can be done by using multivariate probability density functions.
Another consideration when dealing with continuous variables is the issue of infinite probabilities. Since continuous variables can take on an infinite number of values, the probability of any individual value would be zero.
Therefore, we would need to work with ranges or intervals of values instead. However, the basic framework of the Naive Bayes Classifier still remains the same.

III. INTRODUCTION ABOUT LINEAR REGRESSION

Linear regression is a machine learning algorithm used for predicting a continuous numerical output variable based on one or more input variables that have a linear relationship. It assumes that there exists a linear relationship between the input (independent variables) and the output (dependent variable) being predicted. The algorithm determines the best fit line or hyperplane that minimizes the sum of squared differences between the actual and predicted values.
This line or hyperplane can be represented by a mathematical equation in the form of:

$$y = a + bx \tag{5}$$

where y is the predicted output variable, a is the y-intercept, b is the slope, and x is the input variable.
It's possible to have different scenarios of linear regression:

A. One-dimensional linear regression

It is a statistical technique used to model the relationship between a dependent variable and one independent variable. In this regression, there is only one independent variable and one dependent variable. Given a data set, you can distinguish observation **x** and target **t**, mathematically represented as:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ ... \\ x_N \end{bmatrix}, \mathbf{t} = \begin{bmatrix} t_1 \\ t_2 \\ ... \\ t_N \end{bmatrix} \tag{6}$$

A linear model $y(x)$ that predicts t given x is $t \approx y$, where **y = wx**. The goal is to obtain $y(x)$ more similar to $t(x)$ for any x. So, it is trivial to find w, which defined N straight lines (because also w is a vector). The goal is to obtain the best-fitting line (choosing the best w) that minimizes the sum of the squared differences between the predicted values and the actual values. To reach the goal it's necessary to introduce the **Objective Function**:

$$J = \frac{1}{N} \sum_{l=1}^{N} \lambda(y_l, t_l) \tag{7}$$

where: λ is the measurement of the error.
In this particular case the assumption is $\lambda = (y_l - t_l)^2$, where the second term is the square error. So, the (7) becomes:

$$J_{MSE} = \frac{1}{N} \sum_{l=1}^{N} (y_l - t_l)^2 \tag{8}$$

This formula is named **Mean Square Error Objective**. The mean square error (MSE) represents the average squared difference between the predicted and actual values in a dataset. It is commonly used as a measure of the overall quality or accuracy of a predictive model.
Therefore, the wanted w has to be the value which minimizes J_{MSE} and it is computed as the least squares solution to the linear regression problem:

$$w = \frac{\sum_{l=0}^{N} x_l t_l}{\sum_{l=1}^{N} x_l^2} \tag{9}$$

B. One-dimensional linear regression problem with offset

The previous presented model always define a line crossing the origin of the axis, but this kind of model is not always suitable for all situations; for this reason, it's possible to introduce another kind of linear model, whose the mathematically representation is $y = w_0 + xw_1$, where w_0 is called *offset* (or bias) and w_1 *slope* (or gain).
Fundamentally w_0 is the new parameter which causes to switch from a linear model to a model called "affine" and allows the line to cross y-axis in a any point (not mandatory the origin). The more flexible model is now centered around the $\bar{x}$ and $\bar{t}$ computed as:

$$\bar{x} = \frac{1}{N} \sum_{l=1}^{N} (x_l), \tag{10}$$

$$\bar{t} = \frac{1}{N} \sum_{l=1}^{N} (t_l), \tag{11}$$

and the solution for the two parameters is:

$$w_1 = \frac{\sum_{l=1}^{N} (x_l - \bar{x})(t_l - \bar{t})}{\sum_{l=1}^{N} (x_l - \bar{x})^2} \tag{12}$$

$$w_0 = \bar{t} - w_1 \bar{x} \tag{13}$$

C. The multi-dimensional linear regression problem

It is a problem where the data is now composed of d-dimensional vectors and it's possible to define the matrix $\mathbf{X}$ of dimension N x d:

$$X = \begin{bmatrix} x_1 \\ x_2 \\ ... \\ x_N \end{bmatrix} = \begin{bmatrix} x_{1,1}, & x_{1,2}, & ..., & x_{1,N} \\ x_{2,1}, & x_{2,2}, & ..., & x_{2,N} \\ ... & ... & ... & ... \\ x_{N,1}, & x_{N,2}, & ..., & x_{N,N} \end{bmatrix} \tag{14}$$

Since the data are now d-dimensional, we have d parameters in a d-dimensional vector:

$$w = \begin{bmatrix} w_1 \\ w_2 \\ ... \\ w_d \end{bmatrix} \tag{15}$$

and what results is:

$$y = \begin{bmatrix} y_1 \\ y_2 \\ ... \\ y_N \end{bmatrix} = \begin{bmatrix} x_{1,1}w_1 + x_{1,2}w_2 + ... + x_{1,d}w_d \\ x_{2,1}w_1 + x_{2,2}w_2 + ... + x_{2,d}w_d \\ ..... \\ x_{N,1}w_1 + x_{N,2}w_2 + ... + x_{N,d}w_d \end{bmatrix} = X\mathbf{w} \tag{16}$$

The solution of w, always trying to minimize the J_{MSE}, is:

$$w = (X^T X)^{-1} X^T \mathbf{t} \tag{17}$$

where $(X^T X)^{-1} X^T$ is called *Moore-Penrose pseudoinverse of X*.

D. ASSIGNMENT 2

The tasks are:

1) Get data
2) Fit a linear regression model
3) Test regression model

E. Get Data

The provided data set consists of two distinct sets of data. The first data set pertains to the Turkish stock exchange, containing 54 rows and 2 columns. The second data set focuses on MT cars and contains 32 rows and 4 columns. In order to work with the data in Matlab, it was necessary to remove the first column and the first row, which represented the features of the MT cars data set. Matlab specifically operates on numerical values within matrices, hence this adjustment was essential. Both data sets were originally stored in *.csv* files, and they were loaded into separate matrices using the "load" command.

F. Fit a linear regression model

1) One-dimensional problem without intercept on the Turkish stock exchange data: This kind of problem was solved using (9) and the result is represented in this plot:

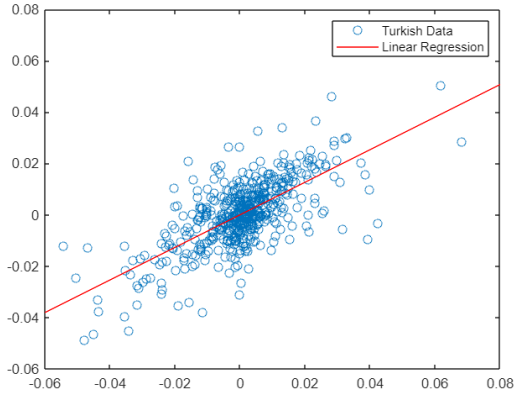


Fig. 3: One-dimensional problem without intercept on the Turkish stock exchange data

2) Compare graphically the solution obtained on different random subsets (10 %) of the whole data set: It's necessary to extract randomly the 10% of the whole data set using the function *randperm*, compute w using (9) and then plot in the same graph the computed data set and the founded model. Then, it is used a *for loop* that allows to repeat this code 10 times (for example) and compare graphically the 10 graphs. Every time you run the code the results changes because of the random data sets and 4 shows an example:

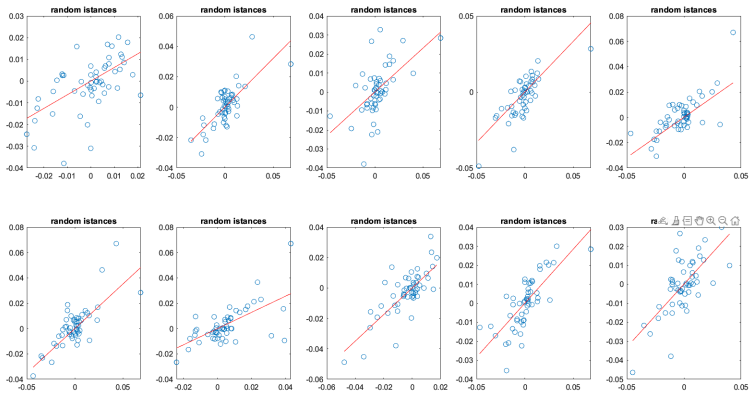


Fig. 4: Comparison between 10 random instances

Also in the previous figure, it's possible to apply the same legend of the 3.

3) One-dimensional problem with intercept on the Motor Trends car data, using columns mpg and weight: Firstly, the computation of $\bar{x}$ and $\bar{t}$ are made and then (13) and (12) are used to compute the bias and the gain of the linear regression.

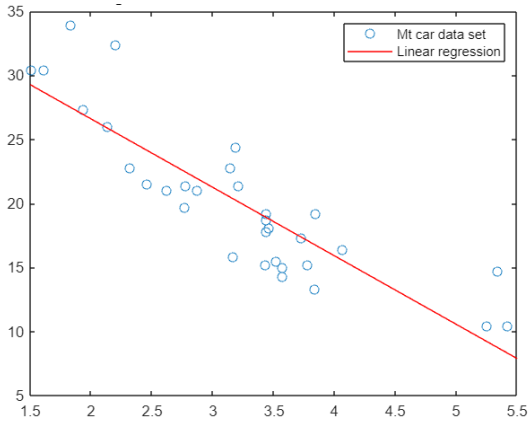


Fig. 5: One-dimensional problem with intercept on the Motor Trends car data

4) Multi-dimensional problem on the complete MTcars data, using all four columns (predict mpg with the other three columns): The first step is to build the matrix, called in the code X_m, where the first column (mpg) is substituted with a column of numbers 1. The "mpg" is the target and it is allocated in a variable called "targ" to use directly its values. At the end, w is computed by using (17) and results a vector of 4 values (in general w must be a vector of $d+1$ parameters **X**.

G. Test regression model

1) One-dimensional linear regression problem with offset, considering only 5% and then 95% of the data: The same code written for the the previous one-dimensional linear regression problem is now adapted here: at the beginning of the script 5% of the data are computed and are used to compute w.

```
% Extract 5% of the whole turkish-se-SP500vsMSCI data set
N = size(turk,1);
allIndices = randperm(N);
m = 5/100 * N;
randomSubset = allIndices(1:m);
matrix1 = turk(randomSubset,:);
```

Fig. 6: How to compute 5% of data

Obviously, every time you run the code the result change. After that, the w, obtained at the previous step, is used to test the model on the remaining 95% of the data. The code used to create the test matrix is the following and it is based on two *for loops*, which scroll and compare all rows of the original matrix and the train matrix (in this case, called *matrix1*) and put in the test matrix ($matrix\_test1$) all the rows not contained in $matrix1$.

```
% Creation of matrix_test1
matrix_test1 = [];
for i = 1:size(turk,1)
    flag = false;
    for j = 1:size(matrix1,1)
        if isequal(turk(i,:), matrix1(j,:))
            flag = true;
            break;
        end
    end

    if ~flag
        matrix_test1 = [matrix_test1; turk(i,:)];
    end
end
```

Fig. 7: How to compute 95% of data

To understand how precise is the model, the Mean Square Error is introduced and computed on both train (J_{mse10}) and test (J_{mse11}), as it's possible to see in 8.

	J_mse10_values	J_mse11_values
1	5.3661e-05	1.1284e-04
2	8.8468e-05	1.0240e-04
3	1.8259e-04	8.4231e-05
4	1.0811e-04	9.3725e-05
5	7.6005e-05	9.1842e-05
6	7.1859e-05	9.5975e-05
7	7.5568e-05	8.9745e-05
8	8.5572e-05	9.7411e-05
9	6.3579e-05	9.4117e-05
10	8.7793e-05	1.0093e-04

Fig. 8: Results on 5% and 95% of the data

It's trivial to notice that MSE values on the train test are close to 0, due to the fact that the model is built on that data. Also the MSE values on test test set are small, so it means the model fits well to data.

2) One-dimensional problem with intercept on the Motor Trends car data, using columns mpg and weight, considering only 5% and then 95% of the data: The same previous considerations can be done for this problem. The summarize of the obtained results is the following:

	J_mse30_values	J_mse31_values
1	0	12.0061
2	0	351.0998
3	5.0487e-29	25.1649
4	0	18.3973
5	0	12.4403
6	3.1554e-29	348.3586
7	1.2622e-29	289.5326
8	2.0510e-29	11.0760
9	3.1554e-29	10.3152
10	0	18.2831

Fig. 9: Results on 5% and 95% of the data

The MSE values on train set are very small, also 0 sometimes; in fact the 5% of the data is only one point (because the MT Car's matrix has 32 row and $0,5 \times 32 = 1,6$). For this reason, during the extraction of data, the function *ceil* (as we can see in 9 was used to consider two points (because the goal is to build a linear model and to define a line two points are necessary).

```
% Extract 5% of the whole mtcarsdata-4features data set
N1 = size(mtscar,1);
allIndices = randperm(N1);
m = ceil(5/100 * N1); % ceil is used to consider 2 points
randomSubset = allIndices(1:m);
matrix3 = mtscar(randomSubset,:);
```

Fig. 10: Use of ceil

Unfortunately, the MSE values on test set are a bigger, but still acceptable in most cases.

3) Multi-dimensional problem on the complete MTcars data, using all four columns (predict mpg with the other three columns), considering only 5% and then 95% of the data: Doing the same considerations of the previous scenario, it's possible to compute the MSE values and the next figure shows the results:

	MSEtrain_values	MSEtest_values
1	5.6407e-26	346.4770
2	5.5918e-25	8.5126e+03
3	9.1271e-27	437.6589
4	8.9173e-27	530.7450
5	9.1271e-27	437.6589
6	1.8093e-25	5.1070e+03
7	1.5462e-27	193.0384
8	2.3350e-28	248.4779
9	3.2817e-27	305.3802
10	2.8525e-27	847.8133

Fig. 11: Results on 5% and 95% of the data

As you can see from the second columns, the model founded does not fit well to the data.
What is interesting to notice are the steps involved to compute the MSE: in this last case, it's firstly necessary to build a new matrix, named in the code Xm and then compute the pseudo-inverse of a matrix.

```
Xm = [ones(size(matrix4,1),1), matrix4(: , 2:4)];
targ = matrix4(:,1);

% Moore-Penrose pseudo inverse
w_multi = (pinv(Xm' * Xm)) * (Xm') * targ;
y_multi = Xm * w_multi;

% mse on train set
MSEtrain = sum((targ - y_multi).^2) / size(matrix4,1);
```

Fig. 12: Computation of MSE in multi-dimensional problem

IV. Conclusion about Linear Regression

The results demonstrate that a linear regression is a straightforward and easily comprehensible model to construct. This can be observed through the analysis of 6 and 7, wherein a well-fitting model is created even with a relatively small data set. Furthermore, by altering the percentage of data considered, it becomes evident that the quality of the model adjusts accordingly. These conclusions are derived from the Mean Square Error calculations.
The goal of this assignment is to comprehend the meaning of variables in the formulas and to adapt them appropriately to the provided data. It is advantageous to validate the outcomes obtained through MSE calculations by employing diverse algorithms to compute them.
The tables presented in all the previous figures are generated using the Matlab software, and a *for loop* is employed to aggregate all the results. It is worth noting that the code execution is repeated 10 times to produce the desired outcomes.

V. Introduction about K-Nearest Neighbors Classifier

A. KNN classifier

The K-Nearest Neighbors (KNN) classifier is a non-parametric machine learning algorithm: it builds a discrimination rule from data, without assumptions about probability distributions. It is based on the principle of similarity. KNN works by identifying the k-nearest neighbors of a given data point in a training data set and then assigning a label to the new data point based on the majority vote from its neighbors. Practically, the KNN classifier calculates the distance between the input data point and all the instances in the training data set. The most common distance metric used is Euclidean distance. Then the KNN algorithm selects the k nearest neighbors based on the shortest distances.
After identifying the k nearest neighbors, the KNN classifier assigns a label to the new data point based on the majority vote of its neighbors. In other words, the KNN classifier counts the number of neighbors belonging to each class and assigns the label that occurs most frequently. In the case of ties, a random selection can be made or it's possible to implement the rule: "k should not be divisible by the number of classes".
The following figure shows in a simple way how the classifier works:

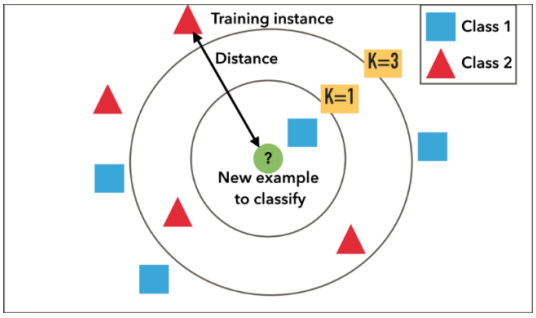


Fig. 13: KNN classifier

13 is taken by Pytholabs 2020.

It's important to note that choosing the value of k is crucial in KNN classification. A small value of k can lead to overfitting, where noise in the training data set is overemphasized. On the other hand, a large value of k can result in underfitting, where important patterns in the data are overlooked.
One advantage of the KNN classifier is that it does not make any assumptions about the underlying data distribution. Therefore, it can handle non-linear decision boundaries and can work well with data sets that have complex and big structures. However, the main drawback of KNN is its computational cost, as it requires calculating the distances between the new data point and all instances in the training data set. This can be computationally expensive, especially for large data sets.

B. Confusion Matrix

The confusion matrix is a powerful tool for assessing (and visualizing) the performance of a classification model by quantifying the number of correct and incorrect predictions made across different classes. It provides a comprehensive summary of how well the model predicts classes or categories. The confusion matrix presents the true positive (TP), true negative (TN), false positive (FP), and false negative (FN) values. These values are organized into a grid-like matrix, where the rows represent the actual or true classes and the columns represent the predicted classes.
The TP value indicates the number of instances that were correctly classified as positive. The TN value represents the number of instances that were correctly classified as negative. The FP (also named type I error) value indicates the number of instances that were incorrectly classified as positive. Finally, the FN (also named type II error) value represents the number of instances that were incorrectly classified as negative.

C. Quality indices

Usually the main indices to understand how a classifier fits well the data are the *accuracy* and *error rate*. Accuracy refers to the measure of correctness in a given context, usually presented as a percentage, and it shows the ratio of true results to the total outcomes. Its opposite is the error rate, which is the proportion of incorrect predictions. It is typically computed as the complement of accuracy, i.e., subtracting the accuracy from 100%. Error rate considers false results (both false positives and false negatives) and represents the amount of mistakes. It quantifies the level of inaccuracy of a classifier.
However, these two indices are not enough to evaluate a classifier and it's necessary to introduce **Quality indices**. These indicators assess accuracy, precision, sensitivity, and the F measurement.

Accuracy calculates the proportion of correctly classified instances by the classifier in relation to the total number of instances. It provides a general overview of the classifier's performance by gauging the overall correctness of its predictions.
Precision and *Sensitivity* are two complementary quality indices that assess the effectiveness of the classifier in identifying positive instances within the data set. Precision measures the proportion of true positive classifications among all positive predictions made by the classifier. It quantifies how well the model avoids false positive errors. On the other hand, sensitivity computes the proportion of true positive classifications relative to the actual number of positive instances in the data set. It reflects the ability of the classifier to avoid false negatives by correctly capturing all positive instances.
The *F measurement* is a quality index that combines precision and sensitivity into a single metric. It is the harmonic mean of precision and recall and offers a balanced evaluation of a classifier's performance. It is a values between 0 and 1 (good model).
Overall, quality indices provide objective measures to assess the performance of classifiers, enabling researchers and practitioners to evaluate and compare different models effectively.

D. ASSIGNMENT 3

The tasks are:

1) Obtain a data set
2) Build a kNN classifier
3) Test the kNN classifier

E. Obtain a data set

The function *Load MNIST* allows to get the data. MNIST is a popular data set used for training machine learning models to recognize digits. First, it's necessary to download the MNIST data set given. The data set consists of two training set files and two test set files. It's important to save these files in a directory accessible by MATLAB to use them.
Here there is the code using to import the data:

```
% Load data
[training_set, target_train] = loadMNIST(0);
[test_set, target_test] = loadMNIST(1);
```

Fig. 14: How to import data set

The numbers 0 and 1 correspond respectively to test and train set.
To avoid long execution time, the function *creation_subset* is created to extract a percentage (in this 5%) of 2 matrices. The code of this function is similar to other assignments.

F. Build a kNN classifiers

The function named *knn_classifier* implements the requested classifier. It takes 4 parameters:

1) *training_matrix*, as a $n \times d$ matrix, to be used as the training set;
2) *target_train*, a corresponding column (a $n \times 1$ matrix) of targets;
3) *test_matrix*, another set of data, as a $m \times d$ matrix, to be used as the test set;
4) *k*, an integer corresponding to the number of nearest neighbors that are considered to classify a new data instance;
5) *target test*, as a $m \times 1$ matrix, to be used as the test set ground truth (this is the no mandatory input).

Firstly, some checks about the matrices given in input are made: the number of arguments received (nargin) must be equal at least the number of mandatory arguments (in this case 4), the number of columns of the second matrix must be equal to the number of columns of the first matrix and k is a number between 1 and the cardinality of the training set, otherwise there are not enough neighbors.
Then, there is the computation of the distances between points belonging to the test matrix and the train matrix. This is done using the function *pdist2*, which takes two input matrices, where each row represents a data point, and computes the pairwise distances between all points in the two matrices. The next function used is *mink*: it finds the k smallest elements in a matrix, specifying (setting the third parameter) the dimension along which you want to find the smallest element. It's possible to extract the target values corresponding to the indices of the nearest neighbors and making predictions based on the nearest neighbors. The code uses the mode function, which finds the most frequently occurring value along each row in the nearest k matrix. The condition $nargin == 4$ checks if there are four input arguments, and if true, it prints the prediction. If there are five input arguments, the code does the same.

G. Test the kNN classifier

In addition to classification, the previously described function also allows returning quality indices as results, obtained using the structure of confusion matrix.

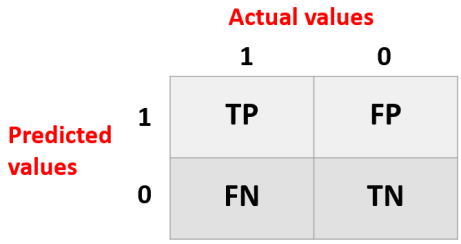
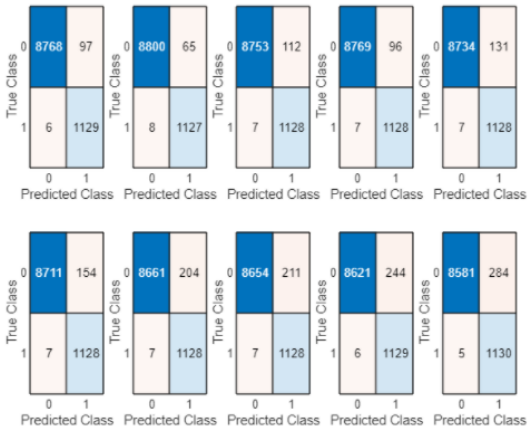


Fig. 15: Structure of confusion matrix

15 was taken by Deepanshi 2023.

To get the confusion matrix, it's possible to use the Matlab function called *confusionmat*.
The code also performs k-Nearest Neighbor (k-NN) classification for different values of k. It calculates and displays various performance metrics for each class and for each value of k, including accuracy, error percentage, precision, sensitivity, specificity, and F measure.
The code begins by setting the different values of k (1, 2, 3, 4, 5, 10, 15, 20, 30, and 40) to be used in the k-NN algorithm. It then initializes variables to store the results for each class and each value of k. A loop is then executed for each class (i) from 1 to 10. Within this loop, the training target is modified to only include values equal to i, and the test target is also modified to only include values equal to i. This ensures that the k-NN algorithm is trained and tested specifically for the current class. For each value of k, the compute kNN function is called to perform k-NN classification on the training and test sets. The resulting predicted class, accuracy, error percentage, precision, sensitivity, specificity, and F measure are stored in corresponding variables. A confusion matrix plot is then generated for each value of k, showing the predicted class labels compared to the actual test class labels.
After the loops have completed, the results, error percentages, precision, sensitivity, specificity, and F measures for each class and each value of k are converted to tables for easy display. The mean and standard deviation of the accuracy, error percentage, precision, sensitivity, specificity, and F measures are computed across all classes for each value of k. These values are also converted to tables for display. Finally, all the results, mean values, and standard deviations are displayed in the command window in a formal and organized manner.

When the code finishes compiling, the output is as follows:



(a) Confusion Matrices

```
Error percent: 1.030000
Accuracy percent: 98.970000
Precision percent: 99.931616
Sensitivity percent: 98.905809
Specificity percent: 99.471366
F measure: 0.994161
```

(b) Quality Indices

Accuracy:

	K = 1	K = 2	K = 3	K = 4	K = 5	K = 10	K = 15	K = 20	K = 30	K = 40
Class = 1	98.97	99.27	98.81	98.97	98.62	98.39	97.89	97.82	97.5	97.11
Class = 2	98.7	98.05	98.65	98.22	98.47	97.93	97.82	97.37	97.01	96.42
Class = 3	98.25	98.2	98.48	98.26	98.5	98.29	98.26	98.16	97.99	97.77
Class = 4	98.24	97.67	98.4	97.95	98.37	98.08	98.14	97.85	97.47	97.18
Class = 5	98.22	98.09	98.32	98.28	98.38	98.14	98.02	97.59	97.19	96.7
Class = 6	99.17	99.27	99.29	99.32	99.27	99.18	99.15	99.03	98.95	98.87
Class = 7	98.15	98.22	98.31	98.32	98.31	98.11	98.06	98	97.9	97.75
Class = 8	98.04	97.46	98.11	97.58	97.96	97.47	97.45	97.1	96.8	96.42
Class = 9	97.44	97.66	97.78	97.93	97.81	97.88	97.63	97.53	97.33	97.17
Class = 10	99.26	99.36	99.31	99.43	99.32	99.23	99.11	99.09	99	98.91

(c) Accuracy for each class and each value of k

Fig. 16: Results

Compiling the code, it's possible to notice, as it was already said, that the code works on subsets, quality indices are computed for each confusion matrices and the table in figure (c) is computed for each quality index. The results depend on the random sampling; therefore, it is necessary to do some statistics over some repeated experiments (in this case 10 different random subsamplings).
In addition, the code shows the mean and the standard deviation of the quality indexes in different tables:

Accuracy:

	K = 1	K = 2	K = 3	K = 4	K = 5	K = 10	K = 15	K = 20	K = 30	K = 40
Class = 1	98.97	99.27	98.81	98.97	98.62	98.39	97.89	97.82	97.5	97.11
Class = 2	98.7	98.05	98.65	98.22	98.47	97.93	97.82	97.37	97.01	96.42
Class = 3	98.25	98.2	98.48	98.26	98.5	98.29	98.26	98.16	97.99	97.77
Class = 4	98.24	97.67	98.4	97.95	98.37	98.08	98.14	97.85	97.47	97.18
Class = 5	98.22	98.09	98.32	98.28	98.38	98.14	98.02	97.59	97.19	96.7
Class = 6	99.17	99.27	99.29	99.32	99.27	99.18	99.15	99.03	98.95	98.87
Class = 7	98.15	98.22	98.31	98.32	98.31	98.11	98.06	98	97.9	97.75
Class = 8	98.04	97.46	98.11	97.58	97.96	97.47	97.45	97.1	96.8	96.42
Class = 9	97.44	97.66	97.78	97.93	97.81	97.88	97.63	97.53	97.33	97.17
Class = 10	99.26	99.36	99.31	99.43	99.32	99.23	99.11	99.09	99	98.91

(a) Confusion Matrices

Mean Error Percent:

K = 1	K = 2	K = 3	K = 4	K = 5	K = 10	K = 15	K = 20	K = 30	K = 40
1.556	1.675	1.454	1.574	1.499	1.73	1.847	2.046	2.286	2.57

(b) Quality Indices

The mean and the standard deviation are computed for each experiment to provide an indication of typical value and an appropriate measure of spread.

VI. Conclusion about kNN-classifier

The KNN classifier is a versatile algorithm for classification tasks, making it an efficient tool in various scenarios. Its reliance on data proximity grants it the ability to handle different distributions effectively. However, it should be noted that this approach has a trade-off of computational complexity. When using a smaller training data set and carefully selecting the value of k, the KNN classifier proves to be effective. It fits the data well and achieves commendable performance. Opting for smaller values of k (e.g., 2 or 3) can yield comparable results. Confusion matrices provide reliable visualization of the classifier's prediction capabilities and overall performance. The benefits of the kNN classification algorithm can be summarized as follows:

1) Easy interpretation of predictions;
2) Few involved parameters;
3) No assumption about probability distributions;
4) No training, but computationally expensive for large data sets.

VII. Introduction about Neural Networks

A. Artificial Neural Networks

Neural networks, often referred to as artificial neural networks (ANNs), are computational models inspired by the result and the inner mechanics of brain processing. These networks consist of numerous interconnected processing elements, called neurons, which work collaboratively to solve complex problems (including classification, nonlinear regression, mapping, sequence forecasting) and make predictions.
The fundamental building block of a neural network is the *neuron*, which takes multiple input signals and combines them using weighted connections determined by numerical parameters called *weights*.

18 shows the structure of a formal neuron:

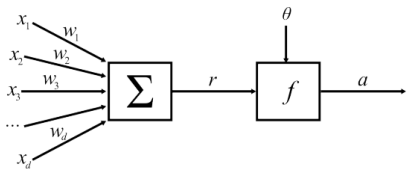


Fig. 18: Formal Neuron

Let's describe all the quantities involved.

1) Weights: Weights (w) are the parameters that determine the strength of the connections between the neurons. They are crucial because they control the flow of signals through the network during the process of forward propagation. Each connection between two neurons has an associated weight that represents the importance or contribution of the sending neuron's output to the receiving neuron. The weights are typically initialized randomly at the beginning and then adjusted during the training phase using optimization algorithms, like Least Means Squares algorithm. The main objective of the LMS algorithm is to minimize the error between the predicted output of the system and the actual output. The algorithm iteratively updates the system weights so that the error gradually decreases, in fact the neural network's aim during training is to learn the optimal set of weights that minimizes the difference between the predicted output and the actual output, known as the loss function.

2) Activation Functions: The weighted inputs are then processed using an *activation function* (f), which generates an output signal. They introduce non-linearity into the model, allowing it to learn complex patterns and make accurate predictions. One of the most common used function is the sigmoid. The sigmoid function maps the input to a value between 0 and 1, making it suitable for binary classification tasks.

Fig. 19: Example of sigmoid function

19 is taken by Wikipedia 2023.

3) Layers: Neural networks are commonly organized into layers, where each layer consists of a group of neurons. The input layer receives the raw data, which is then passed through hidden layers before reaching the output layer. Each hidden layer progressively extracts higher-level features, allowing the network to progressively learn increasingly complex representations of the input. This hierarchical structure is the main point which allows to solve problems. Layers are usually made of homogeneous units, but it's not mandatory they are homogeneous to each other.
The following pictures shows a multi-layer perceptron, also called *shallow network* (when it has only one hidden layer between the input and output layers):

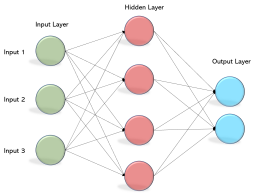


Fig. 20: Multi-Layer Percepton

20 is taken by Camp 2021.

B. Autoencoder

An autoencoder is a specific multi-layer perceptron characterized by two features:

- *Structure*: number of units in the input layer. It is equal to the number of units in the output layer and also greater than the number of hidden units;
- *Learned Task*: an autoencoder is trained to approximate the identity function.

An autoencoder is defined as *unsupervised* because it does not require labeled data for training. In unsupervised learning, the objective is to learn patterns or representations in the given data without any explicit output labels.

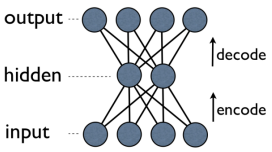


Fig. 21: Structure of an Autoencoder

So, it consists of an encoder network that compresses the input data into a lower-dimensional representation, and a decoder network that reconstructs the original input from the compressed representation. The training objective is to minimize the reconstruction error.

C. ASSIGNMENT 4

In this assignment Matlab's own neural network tools, contained in a library called Neural Networks Toolbox, are used and the required tasks are:

1) Neural networks in Matlab
2) Feedforward multi-layer networks (multi-layer perceptrons)
3) Autoencoder

D. Neural networks in Matlab

The following figure summarizes everything that has been described above:

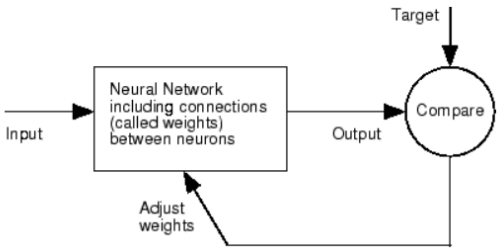


Fig. 22: Train a Neural Network

Deep Learning Toolbox in Matlab is used to accomplish the tasks. Within the toolbox there are different apps. The first one presented is launched by the command *nftool*, which allows data fitting with a shallow neural network and it is also called *Neural Net Fitting*. It contains sample data (e.g., sample body fat data) and it's possible to select and import them. This data can be useful to train a neural network and, in this specific case, to estimate body fat for a subject based on a set of measurements. The information about data are shown in the section called *Model Summary*, which shows the predictors, the responses, the inputs and the targets.
The network is a two-layer feed-forward network with a sigmoidal transfer function in the hidden layer and a linear transfer function in the output layer. The Layer size value defines the number of hidden neurons. The network graph updates based on the input data 23

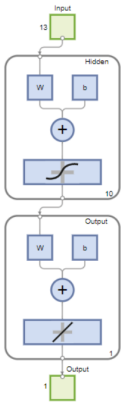


Fig. 23: Scheme of a Neural Network on Matlab

Concerning the training, the training data set is divided in 3 parts to improve the learning of the neural network, but it's possible to change the settings, 24.

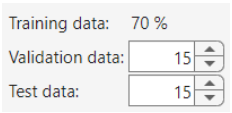


Fig. 24: Division of data set

On this tool, there are different ways to train the neural network:

- *Levenberg-Marquardt* is recommended for most problems, because of its reliability and efficiency;
- *Bayesian regularization* may be a better solution for small problems and also if the noise is involved;
- *Scaled Conjugate Gradient* is recommended because it uses gradient calculations that are more memory-efficient than the Jacobian calculations used by the previous listed methods.

In the Training window Matlab exploits the evolution of the training and it stops until at least one of the stop criteria is verified.
To analyze and understand better the results, the tools provides to generate *Summary tables*, *Regression Graphs*, *Error Histograms*.
All this can be done with the graphical interface, but it's possible to generate automatically the code which executes all the steps described before.

E. Feedforward multi-layer networks (multi-layer perceptrons)

Neural networks excel in pattern recognition tasks. The specialized app, Neural Net Pattern Recognition, is specifically designed for such problems. To initialize the app, the command *nprtool* is used. Users can leverage existing data in the tool, such as the Glass Data Set. Similar to previous tasks, the creation of the neural network, training, and result analysis follow a comparable process.
In this case, the Confusion Matrices and ROC curve serve as the most effective representations for understanding the results. The Receiver Operating Characteristic (ROC) curve is a graphical representation of the performance of a binary classification model. It is commonly used in machine learning and statistics to evaluate and compare the trade-off between the model's true positive rate (TPR) and false positive rate (FPR).
The ROC curve plots the TPR on the y-axis against the FPR on the x-axis. TPR, also known as sensitivity or recall, measures the proportion of actual positive cases that are correctly predicted by the model. FPR, on the other hand, quantifies the proportion of actual negative cases that are incorrectly classified as positive by the model.
The closer the ROC curve is to the top-left corner of the plot, the better the model's performance, indicating a higher TPR and a lower FPR. The curve can be summarized by a single

metric called the Area Under the Curve (AUC), which ranges from 0.5 (representing a random classifier) to 1.0 (representing a perfect classifier).

Like before, users have the option to automatically generate the code.

The following are the results obtained by keeping the default settings unchanged (25, 27, 28, 29):

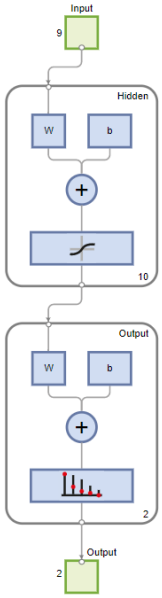


Fig. 25: Neural Network View

Model Summary

Train a neural network to classify predictors into a set of classes.

Data

Predictors: glassInputs - [9x214 double]
Responses: glassTargets - [2x214 double]

glassInputs: double array of 214 observations with 9 features.
glassTargets: double array of 214 observations with 2 classes.

Algorithm

Data division: Random
Training algorithm: Scaled conjugate gradient
Performance: Cross-entropy error

Training Results

Training start time: 27-Dec-2023 10:16:02
Layer size: 10

	Observations	Cross-entropy	Error
Training	150	0.0306	0.0267
Validation	32	0.1192	0.1250
Test	32	0.2867	0.1875

Fig. 26: Model Summary

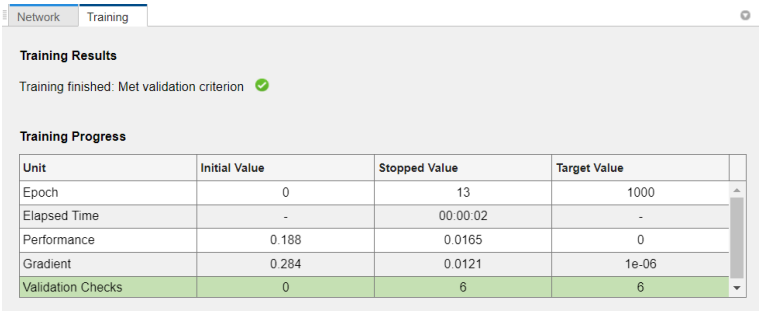


Fig. 27: Training Result

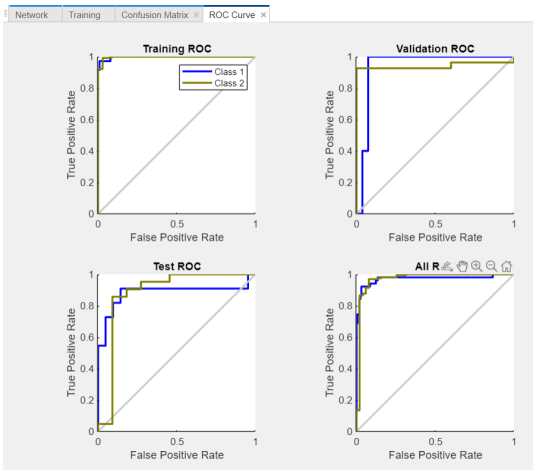


Fig. 28: ROC curve

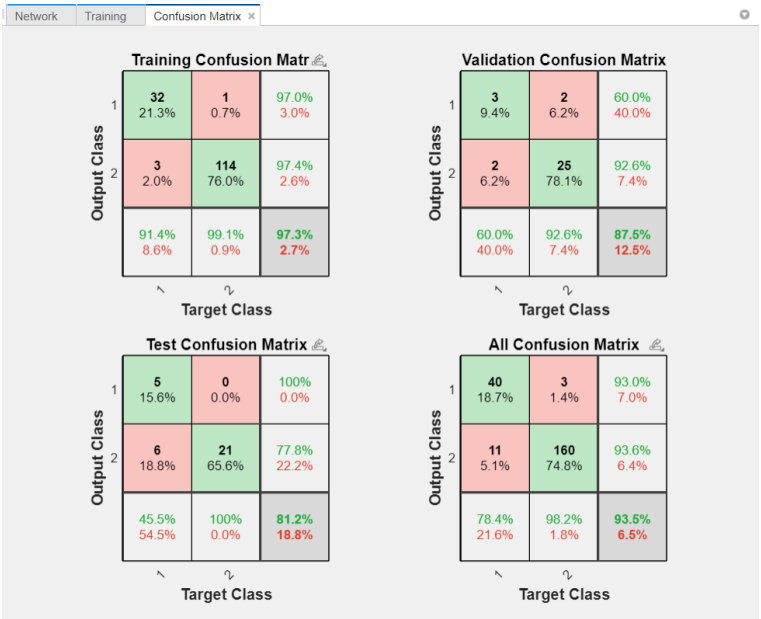


Fig. 29: Confusion Matrices

As already mentioned, some features can be modified, such as the data set division, the number of layers and number of units per layer. In this case, the decision was made to change

the number of layers and the obtained results are presented below.

For instance, in 30 the number of layers is changed from 10 to 30 and it's easy to notice that the values are obviously almost accurate and sometimes they change with respect to the class:

Fig. 30: Confusion Matrices with Layer size=30

If the number of layer is equal to 1, the confusion matrices shows lots of NaN values; that is due to the fact that when setting the size of the layer to 1, one is essentially creating a neural network with only one neuron in the hidden layer. This may result in a limitation in the network's ability to learn and generalize data. If the input data is complex or if the network is too simple, it may be difficult for the model to effectively process information, leading to invalid or indeterminate results in the confusion matrix. So the NN is not very good and as expected it is demonstrated by the ROC curve:

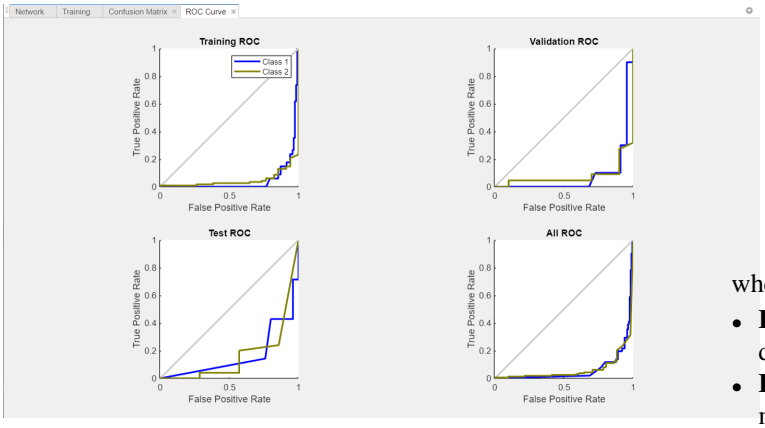


Fig. 31: ROC curve with Layer size=1

F. Autoencoder

The data set used for this task is the same of previous assignment: MNIST. To upload the data set is used the already presented *Load MNIST* function. Firstly, the data set and the targets are stored in the variables *dataset* and *target*, the parameter *nh* is defined equal to 2 and it represents the hidden layers. Then, Matlab offers some functions, *trainAutoencoder*, *encode* and *predict* are use respectively to train, encode and predict the data. It's important to notice that the matrix dataset is transposed, because these functions expect patterns as columns and variables as rows. The results are shown using the function *plotcl* and adding the legend:

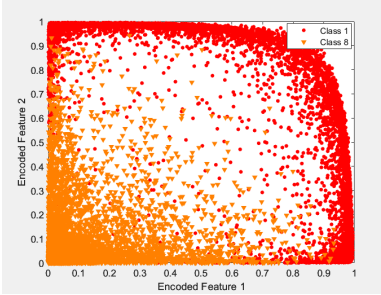


Fig. 32: Plot of the results

In this case, only the classes 1 and 8 are considered.

During the execution of the code (it takes some minutes), the user can follow the evolution of the neural network by the Neural Network Training window:

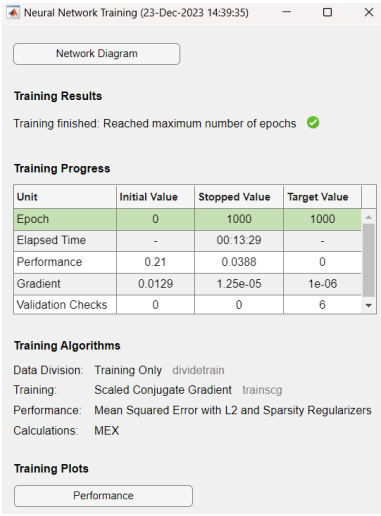


Fig. 33: Neural Network Training window

where:

- **Epoch** represents the number of times the entire training dataset is used to train a model;
- **Elapsed time** indicates the total time taken to train a model using MATLAB's training function. It is displayed to give an idea of how much time it took to train the model.
- **Performance** is a measure of how well the trained model fits the training data. Metrics such as accuracy, precision, or mean squared error are commonly used to evaluate the performance of the model.
- **Gradient** represents the direction and magnitude of the steepest slope of the loss function with respect to the model's parameters.

- **Validation checks** are used to evaluate the model's performance on an independent validation dataset. This validation can help identify iterations of the model that start to overfit or underfit the training data.

All these quantities are considered for the initial value, the stopped value and the target value.
The NN Training window allows to generate the scheme of the autoencoder:

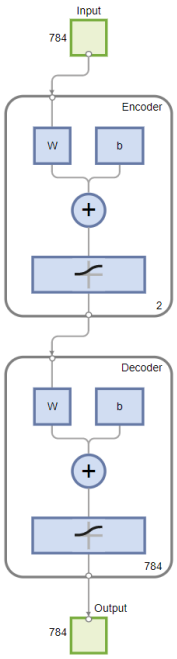


Fig. 34: Autoencoder View

It's possible to recognize the parameter nh as hidden layer and the dimension of the considered data set. The activation function is a sigmoid, but it's obviously possible to change it.

In addition, the NN Training window allows to exploit the performance of the Neural Network:

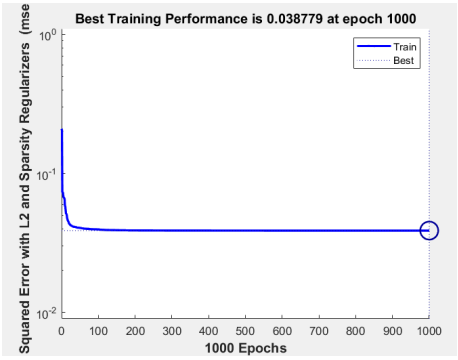


Fig. 35: Performace

The MATLAB graph illustrates the performance trend of a neural network in relation to the number of training epochs. Initially, the performance starts at an initial value ranging from 10^{-1} to 10^{0}. Subsequently, there is a rapid decrease in performance within the first 100 epochs, reaching a value between 10^{-1} and 10^{-2}. After this rapid decrease, the performance stabilizes around that value. This indicates that although the neural network achieved a relatively good performance after the first 100 epochs, the subsequent epochs have a lesser impact on the overall performance as it remains stable around the indicated value, which is also the best one.

VIII. Conclusion about Neural Networks

In conclusion, neural networks have emerged as powerful tools for solving complex problems across various domains. Their ability to learn from data, identify intricate patterns, and make accurate predictions has made them invaluable in today's era of big data. As technology continues to advance, neural networks are likely to play an increasingly pivotal role in shaping the future of artificial intelligence and machine learning.
In this assignment, we explored different aspects of neural networks using Matlab's Neural Networks Toolbox. We used various tools, such as Neural Net Fitting and Neural Net Pattern Recognition, to train and evaluate neural networks for data fitting and pattern recognition tasks.
We learned how to create and train neural networks, adjust their parameters, and analyze their performance using metrics such as confusion matrices and ROC curves. We also explored the concept of autoencoders and their applications in data compression and feature learning.
Through these tasks, we gained hands-on experience with neural networks and deepened our understanding of their inner workings. We understand how different network architectures, activation functions, and training algorithms can impact the performance and accuracy of the models.

References

Camp, Self Study (2021). "Implementing Handwritten Digits Classification from Scratch with Python". In: *https://selfstudycamp.medium.com/implementing-handwritten-digits-classification-from-scratch-with-python-e7daf9f86c94*.

Deepanshi (2023). "In-depth understanding of Confusion Matrix". In: *https://www.analyticsvidhya.com/blog/2021/05/in-depth-understanding-of-confusion-matrix/*.

Pytholabs, Research (2020). "KNN Classifier from Scratch with Numpy — Python". In: *https://medium.com/@lope.ai/knn-classifier-from-scratch-with-numpy-python-5c436e26a228*.

Wikipedia (2023). "Sigmoid Function". In: *https://en.wikipedia.org/wiki/Sigmoid*$_{function}$.