Machine Learning 2021/22

Final Report
EMARO-European Master on Advanced Robotics
DIBRIS - Dip. Informatica, Bioingegneria, Robotica, Ing. dei Sistemi
Università degli Studi di Genova

Matteo Maragliano

January 2, 2022

Introduction

Our work was divided into four different assignments concerning different topics about *Machine Learning* classifiers.
Classification is the process of predicting the class of given data points. Classes are sometimes called in other ways for example targets, labels or categories. Classification predictive modeling is the task of approximating a mapping function f from input variables X to discrete output variables y.
Our assignments concerned the implementation of:

- Naive Bayes Classifier;
- Linear Regression;
- kNN (k-Nearest Neighbour) Classifier;
- Neural Networks.

Each of them was given related to a specific data set to be divided, according to some percentages chosen or given, into training and testing data set in order to create a good classification of the problem. In general, each assignment is divided into different steps, generally three, concerning:

- getting the data set: as specified before, there is a general set to be divided into two smaller ones, a training and a test set;
- building a classifier: here it has to be written some code to create a program which can act as the classifier required;
- testing the classifier: according to the division chosen there is a phase of training the classifier and a phase of testing in which results obtained are compared with respect to the ones expected, also taken as reference during the training phase; in particular the software *learns* from the input and then returns an output according to the learning.

Contents

1 Naive Bayes Classifier **3**
1.1 Introduction 3
1.2 Data Processing 4
1.3 Build a naive Bayes classifier 5
1.4 Improve the classifier with Laplace smoothing 6

2 Linear Regression **8**
2.1 Introduction 8
2.2 Getting the data 9
2.3 Linear Regression Model 9
2.4 Test Regression Model 11

3 kNN Classifier **12**
3.1 Introduction 12
3.2 Obtain a data set 12
3.3 Build a kNN classifier 13
3.4 Test the kNN Classifier 14

4 Neural Networks **16**
4.1 Introduction: Neural Networks in MATLAB 16
4.2 Feedforward multi-layer networks (multi-layer perceptrons) 18
4.3 Autoencoder 20

Chapter 1

Naive Bayes Classifier

The goal of the assignment is to develop a Naive Bayes Classifier to determine possible weather conditions for playing tennis. The first step to be done is the so called Data pre-processing. We are given a data set with a certain number of observations, in particular our data set contains fourteen rows with four columns each one representing an attribute; moreover, there is a fifth column in which is stored the decision, *yes* or *not* in our case, for playing tennis. Our first task consists in converting the given data set into a numeric one so that it can be read by the program used, MATLAB.
We are also given a file containing a brief description of the goal of the project and the characteristics of the data set in order to be able to interpret it.
Once the data set is available for the work we started writing the code.
The project contains two different files: the first implemented is a function which receives as inputs the training data set and the test set and returns as output the target, an array containing the results and the error rate, a number representing the error computed and committed due to the decision taken; the second file is a script in which we use the function, here we load the data set and decide the percentage of rows being the training set and the test set: a common percentage is 70% of the total for the training set while the remaining 30% for the test set, both are taken randomly.
The second task concerns the building of the Naive Bayes Classifier: the task is divided in two phases, one of training and one of testing. All of this can be done by passing through a previous check of the matrices dimensions.
We calculated the probabilities for the different combinations found in the test set and then we fill a cell array with the results.
At the end, the third task to be implemented is the improvement of the classifier with Laplace additive smoothing in order to calculate the probability of observing a value knowing the possible outcomes from other observations.
The last result we obtained is the error rate, calculated as an error among all the possibilities in order to see how far we are in the different conditions processed.

1.1 Introduction

Naive Bayes relies on an assumption that is rarely valid in practical learning problems: that the attributes used for deriving a prediction are independent of each other, given the predicted value. It has been shown that, for classification problems where the predicted value is categorical, the independence assumption is less restrictive than might be expected. For several practical classification tasks, naive Bayes produces significantly lower error rates than more sophisticated learning schemes, such as ones that learn univariate decision trees.

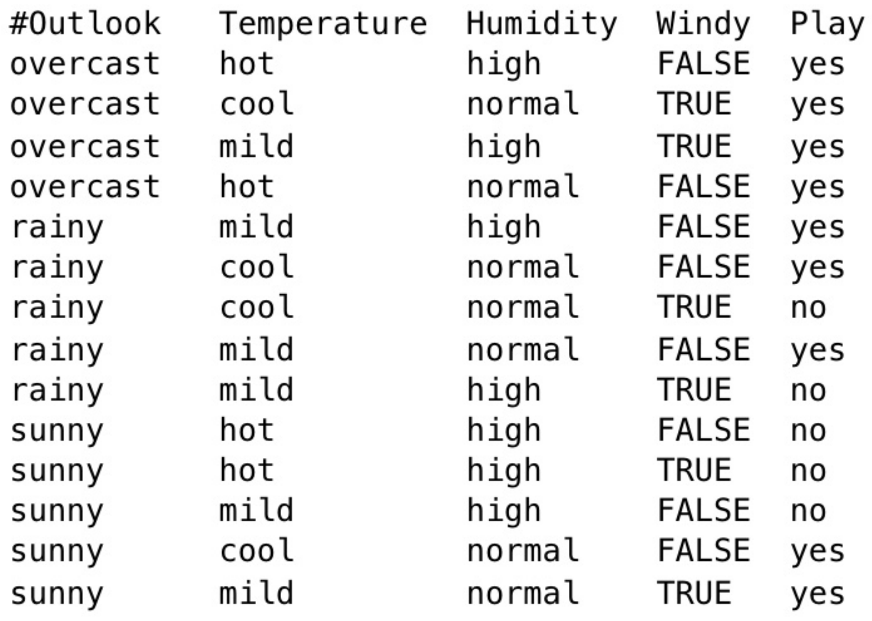
This project has the goal of building a Naive Bayes Classifier starting from an already known data set. A Naive Bayes Classifier is based on the Bayes's theorem, so it works on conditional probabilities: the probability that something will happen, given that something else has already occurred. We can calculate the conditional probability of a certain event using its prior knowledge.

Naive Bayes Classifier predicts membership probabilities for each class such as the probability that a given record or data point belongs to a particular class. The class with the highest probability is considered as the most likely one.

The assumption taken by the Naive Bayes Classifier is that all features are unrelated to each other, so the presence or absence of a particular feature does not affect the presence or absence of another one feature.
We have different classes according to the data set given for the experiment and we want to make a prediction, so we want to know if a certain event will happen or not according to the data.
Since we have a data set we divide it randomly, according to the 70%-30% factor, it is often used in Machine Learning: we take the first percentage as the training set and the other part as the test set. The training set represents all the observations we have and provides us a general vision of the probabilities. Thanks to the training set we can elaborate our decision seeing the test set, so according on what we learned in the first phase of the procedure we can estimate a possible output for a certain combination of features taken from the test set.
At the end of the implementation of the Nave Bayes Classifier we have also implemented a Laplace additive smoothing in order to perform better results on our decisions.

1.2 Data Processing

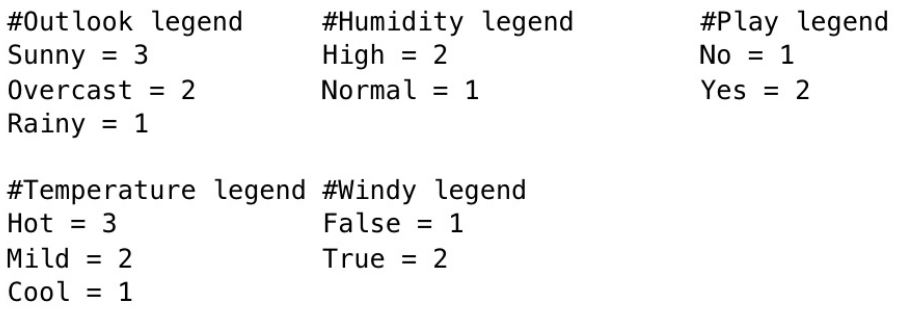
The first task we improved is the so called *Data Processing*. We are given a data set in a written format, as follows:



#Outlook	Temperature	Humidity	Windy	Play
overcast	hot	high	FALSE	yes
overcast	cool	normal	TRUE	yes
overcast	mild	high	TRUE	yes
overcast	hot	normal	FALSE	yes
rainy	mild	high	FALSE	yes
rainy	cool	normal	FALSE	yes
rainy	cool	normal	TRUE	no
rainy	mild	normal	FALSE	yes
rainy	mild	high	TRUE	no
sunny	hot	high	FALSE	no
sunny	hot	high	TRUE	no
sunny	mild	high	FALSE	no
sunny	cool	normal	FALSE	yes
sunny	mild	normal	TRUE	yes

Figure 1.1: Data set given

Since we cannot elaborate literal data we converted them into numerical, according to a legend of only positive numbers: it is not important which value is attributed to each word but it has to be coherent. We chose our legend as follows:

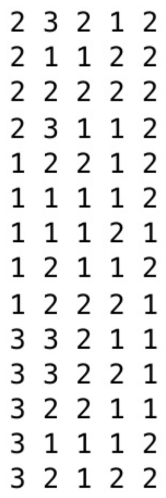


```
#Outlook legend     #Humidity legend        #Play legend
Sunny = 3           High = 2                No = 1
Overcast = 2        Normal = 1              Yes = 2
Rainy = 1

#Temperature legend #Windy legend
Hot = 3             False = 1
Mild = 2            True = 2
Cool = 1
```

Figure 1.2: Legend to convert the data set into a numerical one

Codifying the data set with respect to the legend we obtained a numerical matrix of positive integer numbers:



```
2 3 2 1 2
2 1 1 2 2
2 2 2 2 2
2 3 1 1 2
1 2 2 1 2
1 1 1 1 2
1 1 1 2 1
1 2 1 1 2
1 2 2 2 1
3 3 2 1 1
3 3 2 2 1
3 2 2 1 1
3 1 1 1 2
3 2 1 2 2
```

Figure 1.3: Numerical matrix obtained converting the given data set

Once finished this step we obtained a matrix and so we were able to work on it with the MATLAB software after loading with *load* function.

1.3 Build a naive Bayes classifier

The second task to be solved was the building of the classifier. First of all, before starting with the different phases, we had to check the correct dimensions of both the sets formed, the training set and the test set.
As reference we assumed:

- training set which is a matrix with dimensions *n* x *d+1*;
- test set which is a matrix with dimensions *m* x *c*.

The first check concerned that the number *c* of columns of the test set matrix is at least the number *d* of columns of the training set matrix decreased by *1*.
Once this first check returned a positive result there were two other tests to be performed:

- all the matrices have to have non negative elements: if only one of the values is negative the program returns an error message;
- all the values contained in the test set need to be already contained in the training set: if there was a value which does not respect this rule, the entire row of the test set matrix has to be removed.

Once both sets were ready for being used we started elaborating the data. First of all we computed the number of classes as the number of different values taken from the last column of the training set matrix, we will use this set in the first part.
Having the number of classes, in our case there were two classes represented by the choice *yes* or *not*, we calculated the probability for each variable and each class: we have four variables: Outlook, Temperature, Humidity and Windy and the two classes as already said.
We used the following formulas to calculate our probabilities:

$$P(x_k = true|t_i) = \frac{number x_k = true in class t_i}{tot observation of class t_i} = \frac{N_{true,t_i}}{N_{t_i}}$$

$$P(x_k = false|t_i) = 1 - P(x_k = true|t_i) = \frac{N_{true,t_i}}{N_{t_i}}$$

where we know the prior probability of the class, calculated as:

$$P(t_i) = \frac{number of observation in class t_i}{number of observations in teh training set} = \frac{N_{t_i}}{N}$$

In our MATLAB software we filled a cell array, appropriately initialised with the correct dimensions: it is a matrix in which every location *(i,j)* is an array (there is also the possibility to have different dimension for each array); then we put the calculated probability in each array location.
In our case the cell array was a *(2,4)* matrix in which, according to the different possible values of each attributes, the corresponding arrays have dimension *2* or *3*.

Once the probability cell array is filled properly we have to compute the output of our decision according on the data processed.
In order to make a decision we had to consider the probabilities calculated. We know that every decision made brings a cost and if it is big it means we took a wrong decision.
We measure this cost with a *loss function* $\lambda(y, \omega)$ where y is the decision taken and ω the state of nature. The role of the loss function is to minimise the risk, which will be a conditional risk because based on the observation made. The risk is calculated as follow:

$$R(y_i|\mathbf{x}) = \sum_{j=1}^{c} \lambda(y_j, \omega_j) P(\omega_j|\mathbf{x})$$

and it is the conditional risk of a decision y_i when we have the experimental observation $\mathbf{x}$.
Of course the risk has to be calculated for all the possible cases so, since in this example we are with discrete and categorical variables, we will compute the general risk as:

$$R = \sum_{x \in X} R(y(x)|\mathbf{x}) P(\mathbf{x})$$

where $P(\mathbf{x})$ is the probability mass function of experimental observation and X is the set of all possible inputs (the *input space*).

We compute c blocks $g_j, j = 1...c$, called *discriminant functions*, that computes the total risk.
Since all the variables are independent, the Naive assumption allows us saying $P(x_1, ..., x_d|t_i) = P(x_1|t_i)P(x_2|t_i) \cdot \cdot \cdot P(x_x|t_i)$ so the discriminant functions can be computed as:

$$g_i(\mathbf{x}) = P(t_i)[P(x_1|t_i)P(x_2|t_i) \cdots P(x_d|t_i)] = P(t_i) \prod_{j=1}^{d} P(x_j|t_i)$$

In order to minimise the risk we will choose the minimum value for each decision; consequently, in the software, we will substitute if necessary the value already present in our array with one who takes a lower cost.

Once obtained all the results we also calculated the error rate in order to check how much the decision made by the program deviates from the reality.
The error rate is simply computed by calculating the following formula:

$$errorrate = \frac{numberoftimeswrongdecision}{totaldecisiontaken}$$

1.4 Improve the classifier with Laplace smoothing

The last task to be performed is the addition of a Laplace smoothing in the classifier. This choice is determined by the presence of many zeros in the discriminant functions due to few data available in a small data set.
For that task we had to calculate again the probabilities, in order to avoid having so many zeros; in particular we have a variable x which assumes values in a range *1,2,...,v* with v the possible discrete values. The Laplace probability of observing a specific value i is:

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

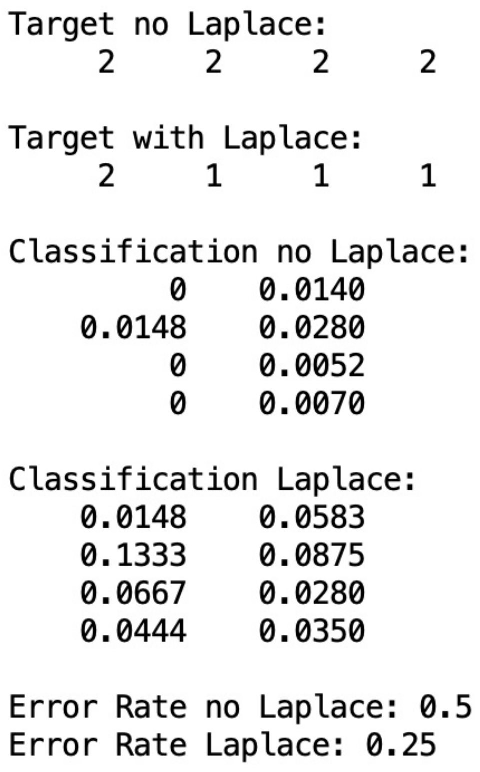
where n_i is the times the value i occurs and a is a parameter properly chosen:

- a>1 means that we trust the prior belief more than the data;
- a<1 means that we trust the prior belief less than the data;

for our purpose we will use a=1.

Of course also for this second set of results we computed, as before, the error rate and then we compare with the previous one calculated. The comparison pointed out that with Laplace smoothing the error rate can decrease in respect to not using it.

As an example have a look at the following figure:



```
Target no Laplace:
     2     2     2     2

Target with Laplace:
     2     1     1     1

Classification no Laplace:
         0    0.0140
    0.0148    0.0280
         0    0.0052
         0    0.0070

Classification Laplace:
    0.0148    0.0583
    0.1333    0.0875
    0.0667    0.0280
    0.0444    0.0350

Error Rate no Laplace: 0.5
Error Rate Laplace: 0.25
```

Figure 1.4: Results taken from a random experiment

The figure points out what we said before. The result of *Classification* represents the *discriminant functions* and we can see that with the use of the Laplace smoothing the zeros can be avoided; moreover the error computed with the use of Laplace smoothing is lower than before so it produces a better result in terms of decisions made: we can see in the *target* that all values calculated differs for the first element in the array, so it means that in that case the decision taken was better.

Chapter 2

Linear Regression

The goal of the assignment is to implement a *linear regression* among a certain amount of data given. *Regression* means approximating a functional dependency based on measured data.
The first task to be implemented concerned getting the data and make them ready to be read and used by the program, in our case MATLAB. We were given two different data sets to be used in different ways: the first one has two columns, the first represents the input and the second the output; the second data set has four different columns representing *mpg (miles per gallon)*, *disp (displacement)*, *hp (horse power)* and *weight* respectively.

Then we had to complete the second task:

- first of all we had to implement a one-dimensional problem without intercept using the first data set;
- then we had to compare it graphically with respect to a random subset of dimension 10% of the total one and find a regression without intercept as well;
- the third point took the second data set and used its first and fourth column, *mpg* and *weigh*t respectively; we had to compute another one-dimensional problem using the *weight* as the output and the *mpg* data as the input;
- the last one point dealt with a multi-dimensional problem using the last three columns of the second data set to predict the mpg, the first one column.

The third task concerned the re-make of the point 1,3 and 4 of the task 2 using this time 5% of the total data given. Then we had to compute the objective, Mean Square Error (MSE), on the training data and also on the remaining 95% of the data.
In conclusion the program had to be tested multiple times, for example 10 times, and then all the results to be shown on a graph or in a table to be discussed properly.

2.1 Introduction

In statistical modeling, regression analysis is a set of statistical processes for estimating the relationships between a dependent variable, often called the *outcome* or *response* variable and one or more independent variables.
The most common form of regression analysis is *linear regression*, in which it has to be found the line, or a more complex linear combination, that most closely fits the data according to a specific mathematical criterion.
For specific mathematical reasons, this allows the researcher to estimate the conditional expectation of the dependent variable when the independent variables take on a given set of values.

Moreover, it can also be used to find some values not inside the range of the independent variable.

2.2 Getting the data

The first task to be completed is the one in which we had to get the data from given data sets and make them readable for out software, MATLAB.
We were given two different data sets, the first one containing only two columns, one for the input and one for the output. In MATLAB we could read it by a specific function, *load.*

With the second data set it was a little be more complicated since we had also a literal columns, the first one, and other three number columns. With the *readmatrix* function we transformed the data set into a matrix and then we proceeded to remove the first column since it was not necessary for us and not readable by our software. So at this time we had all data available and ready to be processed.

2.3 Linear Regression Model

This task concerned the presence of four different points to complete. In the first one we had to model a linear regression of a one dimension problem, using the first data set given. In this case we did not have to consider the intercept of the linear regression, because of the zero mean value of the data set.
We proceeded by calculating the angular coefficient, stored in a vector with proper dimensions, by doing the division of all the outcome values and all the independent values. After that, we could compute the outcome of the regression by simply multiplying the angular coefficient obtained with the x-values of the data set. The formula used is:

$$y = wx$$

All results obtained were plotted in a graph.

The second point was just a comparison between the previous result obtained and an analogous problem computed with a sub set of 10% the dimension of the initial set. All the steps computed were the same and also in this case the result was plotted on a graph.

In the third point of the task the problem was still a one dimensional problem but in this case there was also the intercept to be computed. We proceeded in the same way only slightly changing the computation of X, all the values of the independent variable. In this case the formula for the computation of the outcome is:

$$y = w_1x + w_0$$

where w_0 is the intercept of the line computed as: $w_0 = \bar{t} - w_1\bar{x}$ where $\bar{t}$ is the mean value of the target and $\bar{x}$ the mean value of the input.

Then the angular coefficient and the computation of the outcome were the same as before and the result plotted in a graph gave the result attended.

The most difficult point was the last one because of the presence of a multi dimensional problem; in this case the *target* was the first column whereas the input variables were the other three columns of the data set. Also in this case there was the intercept to be computed.
The multidimensional problem can be modelled as follows:

$$X = \begin{pmatrix} \mathbf{x_1} \\ \mathbf{x_2} \\ \vdots \\ \mathbf{x_N} \end{pmatrix} = \begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,d} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,d} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N,1} & x_{N,2} & \cdots & x_{N,d} \end{pmatrix} \qquad \mathbf{w} = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_d \end{pmatrix} \qquad \mathbf{t} = \begin{pmatrix} t_1 \\ t_2 \\ \vdots \\ t_N \end{pmatrix}$$

The formula used to computed the angular coefficient is a little different by the one used in the previous software. Anyway, nothing changes in terms of result; the theoretical one used is the following:

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

where $\mathbf{t}$ represents the target vector. One computed the $\mathbf{w}$ vector the outcome is as always the multiplication of the input and the angular coefficient.

The graph in this case represents an approximation of the diagonal of the square having side from the minimum to the maximum value: the dot-graph is an approximation whereas the diagonal is the exact diagonal took as reference.

We give a general view of the result with all the following graphs:

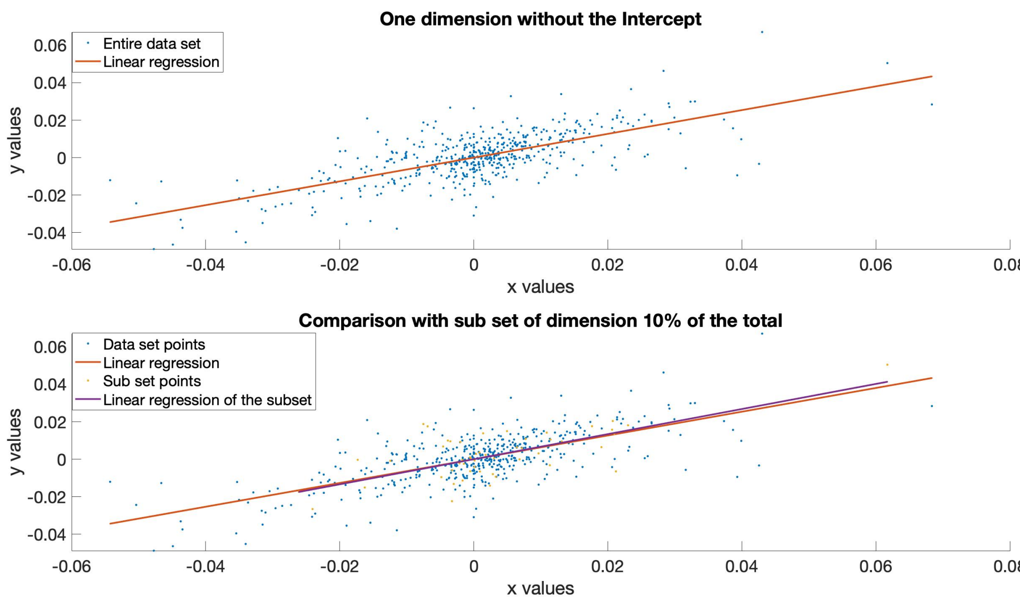


Figure 2.1: Plots concerning the first and the second point of the task: the first is the entire data set; the second represents a comparison with a subset of 10% the dimension of the original one.

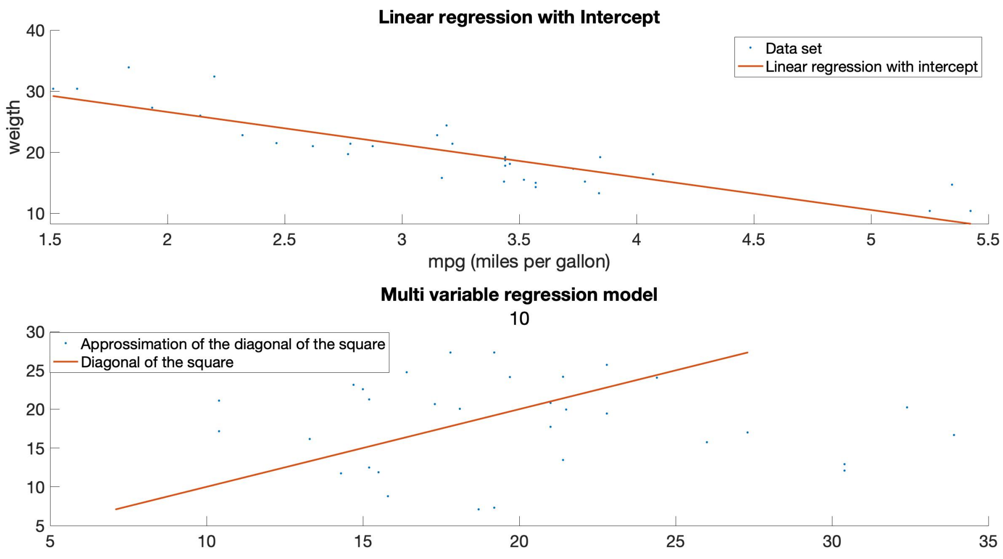


Figure 2.2: Plots concerning the third and fourth point of the task: in the first graph there is the regression from *mpg* to *weight*; the second graph, as it can be seen, represents the approximation of the diagonal and the diagonal itself.

2.4 Test Regression Model

In the last one task, we had to test the regression model obtained by computed errors on two sub sections of the data set.
We started by dividing the set into two different parts: according to a given percentage of 5-10% we had the training test and the remaining 95-90% the test set. Since the division is randomly computed, we had to do this for a certain number of iterations in order to verify the behaviour: for example 10 different iterations.

In order to do the task we computed the angular coefficient with the test set and then, with the value obtained, we computed the outcome by using all the target values of the training set. Then the error was computed by using the *Mean Squared Error (MSE)* among the outcome computed and the given target of the set; of course both done for the test and the training set.
At the end all the results were plotted by a histogram graph to show the error.

We give the plot obtained as follows:

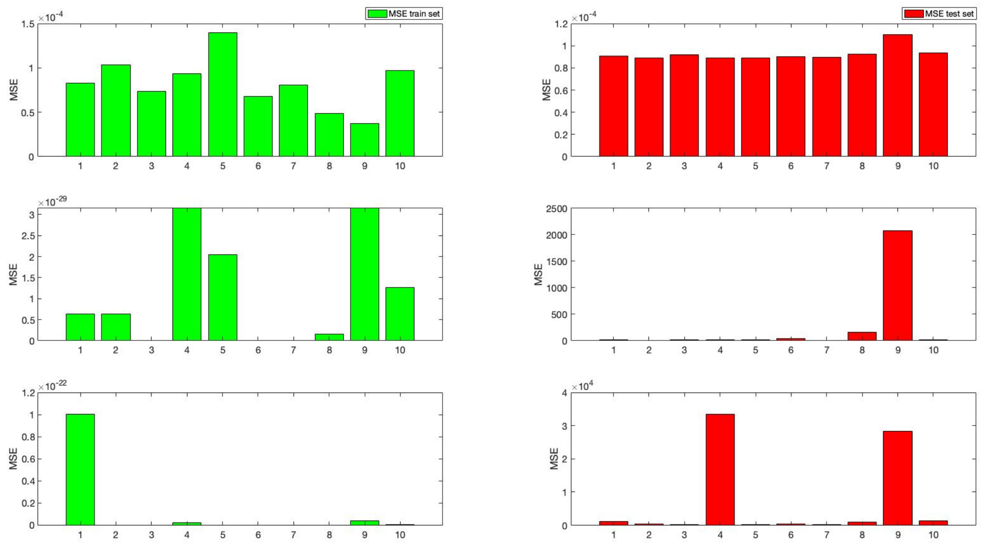


Figure 2.3: Comparison among errors for linear regressions obtained

We plotted the training set with green bars while the test with red ones.
Of course, as it can be seen in the picture, the results of the training set errors are much lower than those of the test set because of the small percentage given to the division: since training errors are values near to zero it is very difficult to see them, in particular in the third case.
Moreover, if we analyse the graph properly it can be seen that for the test set the error values are very high in certain case, this is due to the computation of the output y with the angular coefficient $w1$ used for the training: the small percentage given to the training set brings a high error during the test phase; in our case the training set had only 2 rows so the angular coefficient computed with the regression was the one of the line passing through the two point and, of course, during the test it is not as precise as during the training.

Chapter 3

kNN Classifier

The goal of the assignment is to implement a *kNN Classifier* (k-Nearest Neighbors Classifier) among a certain data given.
In kNN classification, the output is a class membership. An object is classified by a plurality vote of its neighbors, with the object being assigned to the class most common among its k nearest neighbors (k is a positive integer, typically small). If $k = 1$, then the object is simply assigned to the class of that single nearest neighbor.
In particular we were provided different data, from which we took the train and the test set. In particular we pointed out two variables for each data set: X and T representing the matrix of the classes and the labels for each class respectively.

3.1 Introduction

The kNN algorithm is mainly used for classification problems, even if it can also be used in case of regression problems.
The algorithm uses the entire training instances to predict output or unseen data; the model is not learned using training data prior and the learning process is postponed to a time when prediction is requested on the new instance.

3.2 Obtain a data set

As we explained before, the first step to compute is getting data. The function used is *loadMNIST* which takes different parameters as inputs:

- 0 : used to take the X and T for the training set;
- 1 : used to take the X and T for the test set.

There is also the possibility to give a second input: a parameter that describe the type of label to take; in our case digits from 0 to 9 where the 0 represents the 10 while the other are the same digit, so the loading data steps is:

$$[X_{train}, T_{train}] = loadMNIST(0, 0:9);$$
$$[X_{test}, T_{test}] = loadMNIST(1, 0:9);$$

Each item represents a different matrix, in particular:

- Xs are matrices with dimensions:
  - Train: 60000 x 784;
  - Test: 10000 x 784;
- Ts are matrices with dimensions:
  - Train: 60000 x 1;
  - Test: 10000 x 1.

It is useful to know that there is a correlation between the two output data given by the function, in particular each label T is represented by a vector X of 784 elements; these vectors represents figures and they can be shown by typing on the command windows *imshow(reshape($X_{train/test}$(row,:),28,28))*.
We can see it clearly taking the first row of the training set and checking in the corresponding label column the correspondence:
type $imshow(reshape(X_train(1,:),28,28))$ and the output will be:

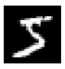


Figure 3.1: Representation of the figure of the first row of the training set

After that, if we checked the corresponding label inside the T_{train} (as we called it) we would see that there is ***5*** which is correct. Of course this can be done for each row either for the training and for the test set.

Another important parameter used is the k whose value represents the number of neighbours to look for for a certain target chosen.

3.3 Build a kNN classifier

This task concerned the building the classifier.
First of all there is the definition of the parameters the function has to take:

- a set of data, as a n x d matrix, to be used as the training set;
- a corresponding column (a n x *1* matrix) of targets;
- another set of data, as a m x d matrix, to be used as the test set;
- an integer k.

First of all the program should verify the correctness of the dimension of the data sets given.
Before starting to implement the total classifier for different value of k, the easiest way is to implement it for $k = 1$.

There are various steps to be processed to have the classifier ready. Once all the dimensions are correctly checked, we computed the Euclidean distances between each row of the train with each row of the test, storing all the results into a matrix with dimensions m x n.
Once filled the matrix, we searched, for each row, the minimum value for the k searched (1 in the first case). The software MATLAB does not give the value itself but the index, so with a simple operation we can take the row with inside the minimal value of the distance.
After that, we computed the mode of the row selected, to determine the most frequent value: this will be our target.
More specifically, we are given a specific data set $X = \{\mathbf{x}_1, ..., \mathbf{x}_l, ..., \mathbf{x}_n\}$ and a *query* point $\bar{\mathbf{x}}$. The general algorithm for a k-Nearest Neighbour is:

$$\{n_1, ..., n_k\} = top_k||\mathbf{x}_l - \bar{\mathbf{x}}||$$

$$y = mode\{t_{n_1}, ..., t_{n_k}\}$$

In case the k required is 1, the y computation changes in:

$$q = argmin||\mathbf{x}_l - \bar{\mathbf{x}}||$$

$$y = t_q$$

At the end we computed the error rate for our result, to determine how far the classifier went from the test: to solve this we compared the target matrix with the test matrix and summed an error for each mistaken value detected.
The error was computed like:

$$err = err + \frac{err}{length(target)}$$

With this formula the error is proportioned to the length of the data set taken.

3.4 Test the kNN Classifier

This section provided the testing of the classifier built before.
First of all, since the data sets given are too wide to compute a result for all the possible combination, we took a subsets:

- 5% for the training set: we trained the classifier with 3000 rows;
- 15% for the test set: we tested the accuracy of our classifier with 1500 rows.

The classifier has been tested for multiple value of k, representing the number of neighbours required, as we were suggested for example $k = 1, 2, 3, 4, 5, 10, 15, 20, 30, 40, 50$.
To better visualise the result we provided a graph in which we plotted on the *x-axes* the different values of k while on the *y-axes* the *accuracy*. The accuracy was computed as:

$$accuracy = 1 - error_{rate}$$

where the *error rate* is a matrix, with properly dimensions, containing all the error committed computed for every row of the test with respect to the target obtained, as said before.

The graph result is the following:

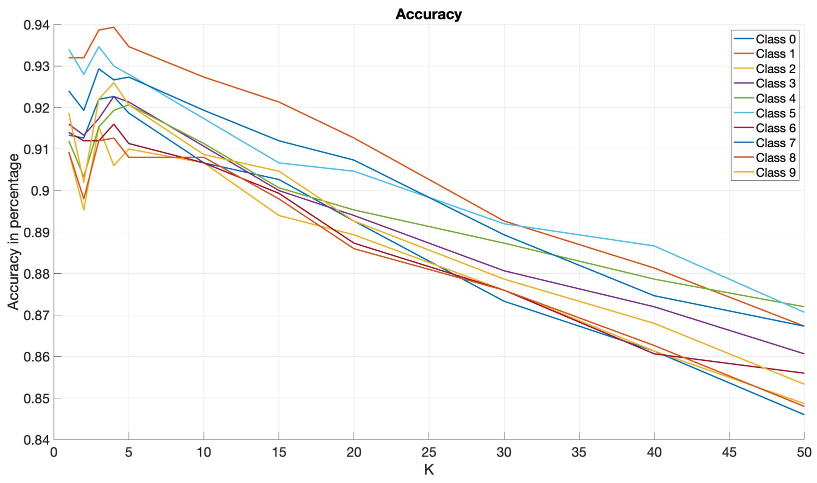


Figure 3.2: Representation of the accuracy of the kNN classifier built.

It can be clearly seen that the accuracy decreases while the number k of neighbor required increases: this is because, with the increasing of the number of neighbours, a higer number of labels satisfied the condition, even if while testing they represent an error.
From the graph it is clear that the accuracy decrease with the increasing of the number of neighbours and the decreasing is, with a good approximation, a linear decrease. In particular there are two situations in which there is an abnormal situation:

- k = 2: we can see the presence of a huge peak down. this behavior is due to an ambiguity in the choice of values, since there are two neighbours the mode is a choice between a 50% of possibility.
- k = 4: as before also in this case there is the possibility of an ambiguity, even if it is less probably since there are more values to analyze; in fact the peak down is less evident than before.

In particular we can say that, already the 25^{th} neighbours is not in the same class as the first one, so with k too high we consider also values not correct for the experiment and so this can brings some errors inside the computation of the output.
This of course depends on the dimension of the data set taken: if we had taken a larger subset or the entire data set we would have had a better result for the accuracy and the error.
Nevertheless the result is quite good since the value does not go below the 80% of accuracy with the highest number of neighbours.

In the following graph it is also represented the error computed:

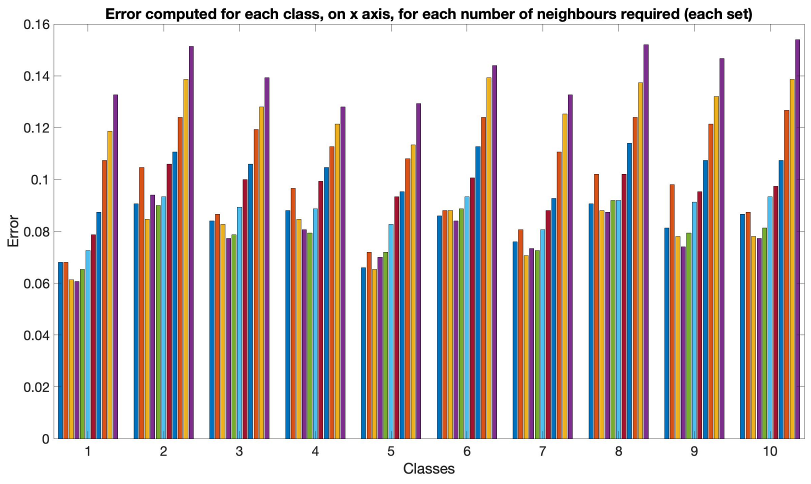


Figure 3.3: Representation of the error computed by the classifier

As it is reasonable, for big ks there is typically a bigger error. The bar graph is the dual to the first one since they are obtained by $1 - error_{rate}$ for the first and $1 - accuracy$ for the second.
Also here we can see the critical behaviour with k=2 and k=4 as before.

Chapter 4

Neural Networks

The goal of the assignment is to test the Neural Network Toolbox of MATLAB. Thanks to this tool we could test and train different types of data sets and see results through *confusion matrices* or *ROC curve: Receiver Operative Characteristics curve* which describe the performances of the work.
Data are often divided into classes: in our case the data set concerns the identification of cancer into benign or malignant using features of sample biopsies: in particular we had 9 inputs each one with 699 different observation and an output with 2 classes, benign or malignant as just said before, with 699 observations as well in order to classify the result.

4.1 Introduction: Neural Networks in MATLAB

Neural networks are networks of several, interconnected, simple units. They are a model for learning and mapping from inputs to outputs that tries to emulate the inner mechanics of brain processing.

The model of neural network used in our work is the following:

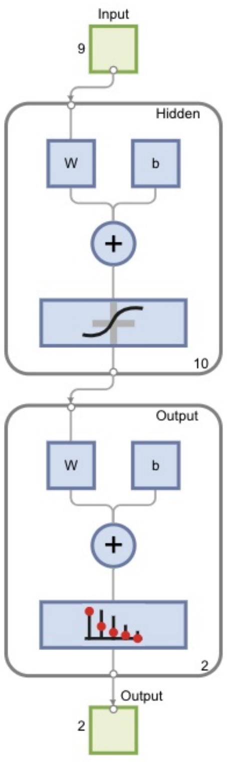


Figure 4.1: Neural network model: as it can be seen in the figure there are 9 inputs (which represent 9 observations) and 2 outputs (which represent the 2 classes returned by the classification).

The basic element of the network is the neuron which is a simple unit that takes an input, makes a decision based on a function and returns an output that can be passed to another neuron or to the network itself. The model of neural function, a formal neuron, is the following:

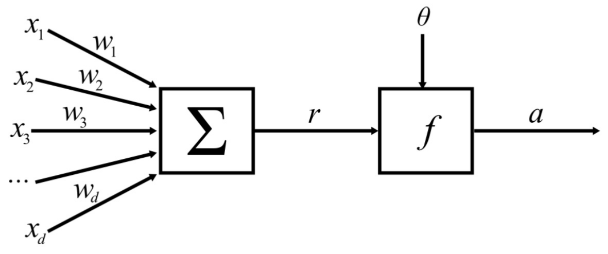


Figure 4.2: Formal Neuron model

Formulas behind neurons are:

$$r = \mathbf{x} * \omega$$
$$f = (r - \theta)$$

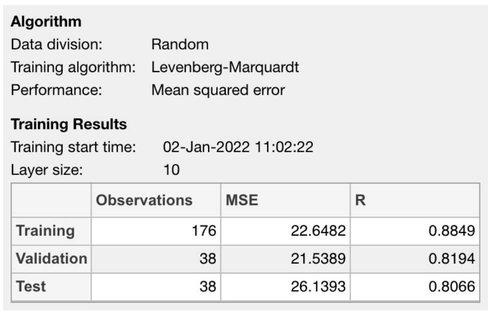
where:

- x is the d-dimensional vector of input to the network and w are the corresponding weights;
- r is the net input on the neuron membrane;
- f is a nonlinear function used to take the decision;
- θ is a threshold;
- a is the output of the neuron (activation value or action potential) and is $a = f(r - \theta)$.

There are various type of functions that can be used: some of the most popular are the *Heaviside Step*, the *signum* and the *sigmoid* represented by the following formula:

$$f(x) = \frac{1}{1 + e^{-x}}$$

In this first task we fitted the neural network by using the *Network Fitting Tool* by typing *nftool* into the command window.
Neural networks are good at fitting functions. In fact, there is proof that a fairly simple neural network can fit any practical function. The network is a two-layer feedforward network with a *sigmoid* 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.
Training with *Levenberg-Marquardt*, as we did, is recommended for most problems. For noisy or small problems, *Bayesian Regularization* can obtain a better solution, at the cost of taking longer. For large problems, *Scaled Conjugate Gradient* is recommended as it uses gradient calculations which are more memory efficient than the Jacobian calculations the other two algorithms use.
Here there are all the results obtained with a 70% of training, 15% for the validation and 15% for the test:



Algorithm

Data division:	Random
Training algorithm:	Levenberg-Marquardt
Performance:	Mean squared error

Training Results

Training start time:	02-Jan-2022 11:02:22
Layer size:	10

	Observations	MSE	R
Training	176	22.6482	0.8849
Validation	38	21.5389	0.8194
Test	38	26.1393	0.8066

Figure 4.3: Model summary with information about the training algorithm and the training result for each set.

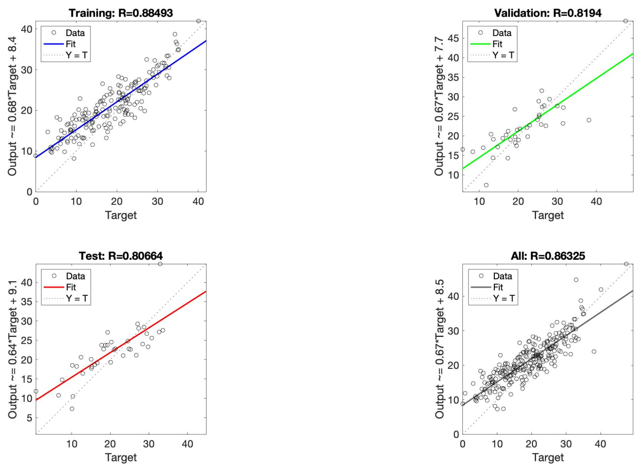


Figure 4.4: Regression for the results obtained

For a perfect fit, the data should fall along a 45 degree line, where the network outputs are equal to the responses. For this problem, the fit is reasonably good for all of the data sets.
We can also check the error histogram to obtain additional verification of network performance:

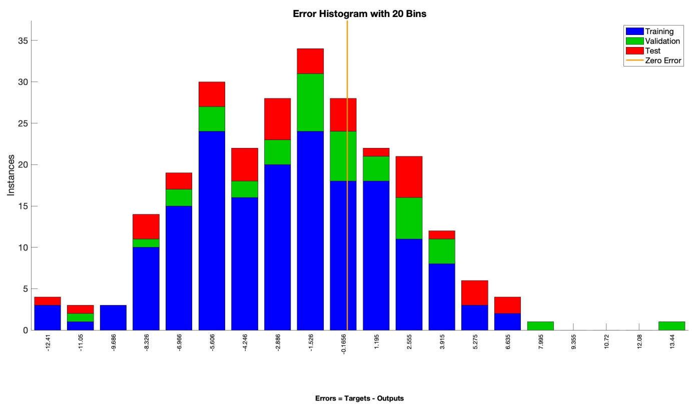


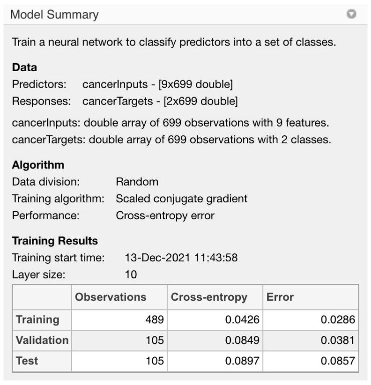
Figure 4.5: Error histogram for the set required

The histogram provides an indication of outliers, which are data points where the fit is significantly worse than most of the data. It is a good idea to check the outliers to determine if the data is poor, or if those data points are different than the rest of the data set. If the outliers are valid data points, but are unlike the rest of the data, then the network is extrapolating for these points.

4.2 Feedforward multi-layer networks (multi-layer perceptrons)

Inside the first task of the assignment we were supposed to learn from an article on how to use a tool.
The tool mentioned is the *Neural Network Fitting app* which is an application able to represent and simulate the behaviour of a neural network.
Thanks to this tool we could have an interface and so a better way to approach the problem. The model of the neural network proposed is the one showed in the figure, in particular the network is a two-layer feedforward network with a *sigmoid* 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.
Using the application we chose a data set from a group proposed and trained the set; after that we had the result of the simulation in different forms: a model summary which contains information about the training algorithm and the training results for each data set; there is also the possibility of checking the results obtained by generating the graphs we want: confusion matrices, regressions, histograms and so

on.
For our example we took the confusion matrices and the regression ones: the results obtained are shown in the figures below.


Model Summary

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

Data
Predictors: cancerInputs - [9x699 double]
Responses: cancerTargets - [2x699 double]

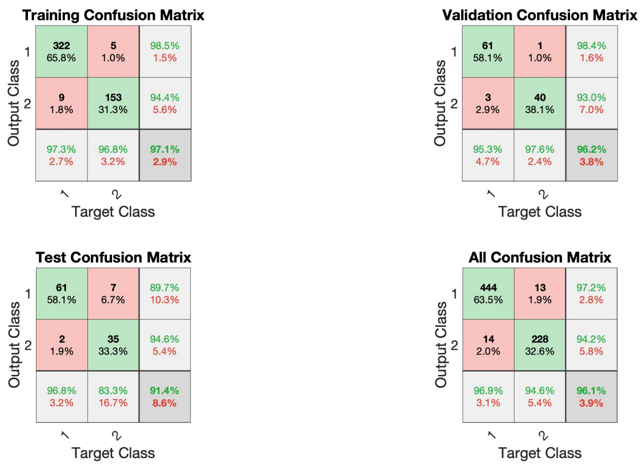
cancerInputs: double array of 699 observations with 9 features.
cancerTargets: double array of 699 observations with 2 classes.

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

Training Results
Training start time: 13-Dec-2021 11:43:58
Layer size: 10

	Observations	Cross-entropy	Error
Training	489	0.0426	0.0286
Validation	105	0.0849	0.0381
Test	105	0.0897	0.0857

Figure 4.6: Model Summary for the Neural Network with 70% of data for training, 15% of data for test and 15% of data for validation.


Training Confusion Matrix

Output Class	1	2	
1	322 65.8%	5 1.0%	98.5% 1.5%
2	9 1.8%	153 31.3%	94.4% 5.6%
	97.3% 2.7%	96.8% 3.2%	97.1% 2.9%

Target Class

Validation Confusion Matrix

Output Class	1	2	
1	61 58.1%	1 1.0%	98.4% 1.6%
2	3 2.9%	40 38.1%	93.0% 7.0%
	95.3% 4.7%	97.6% 2.4%	96.2% 3.8%

Target Class

Test Confusion Matrix

Output Class	1	2	
1	61 58.1%	7 6.7%	89.7% 10.3%
2	2 1.9%	35 33.3%	94.6% 5.4%
	96.8% 3.2%	83.3% 16.7%	91.4% 8.6%

Target Class

All Confusion Matrix

Output Class	1	2	
1	444 63.5%	13 1.9%	97.2% 2.8%
2	14 2.0%	228 32.6%	94.2% 5.8%
	96.9% 3.1%	94.6% 5.4%	96.1% 3.9%

Target Class

Figure 4.7: Confusion Matrices for the Neural Network with 70% of data for training, 15% of data for test and 15% of data for validation.

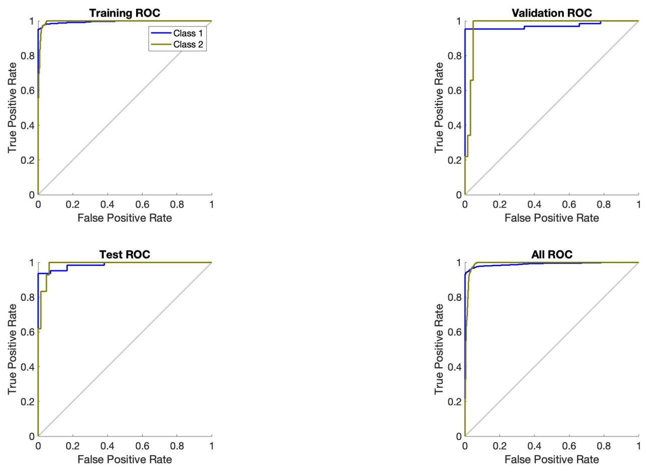


Figure 4.8: ROC curve for the Neural Network with 70% of data for training, 15% of data for test and 15% of data for validation.

From the figure it is evident that the network outputs are very accurate, as it can be seen by the high numbers of correct classifications in the green squares (diagonal) and the low numbers of incorrect classifications in the red squares (off-diagonal).
Also from the other graph is evident what we just said, in particular the colored lines in each axis represent the ROC curves. The ROC curve is a plot of the true positive rate (sensitivity) versus the false positive rate ($1 - specificity$) as the threshold is varied. A perfect test would show points in the upper-left corner, with 100% sensitivity and 100% specificity. For this problem, the network performs very well.

4.3 Autoencoder

To finish our work we completed the task number 2 with the implementation of an *Autoencoder*.
The simplest autoencoder network is a multi-layer perceptron neural network which has one input layer, one hidden layer and one output layer; the model used is the following:

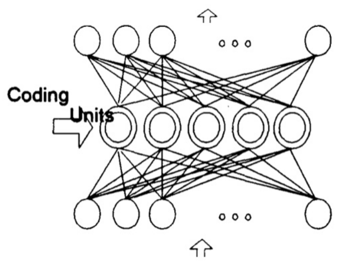


Figure 4.9: Structure for the *Autoencoder* implemented.

It is trained using the same pattern as both the input and the target; note that in this case we don't have any classes or other mapping to learn. This is a special case of *unsupervised training*. In fact, it is sometimes called *self-supervised*, since the target used is the input pattern itself.
An autoencoder learns an internal, compressed representation for the data. The interesting output, therefore, is the value of its hidden layer. What we hope is that similar patterns will have similar representations, in other words, that the network will learn the classes.

To complete this, we gave 2 distinct classes as input and as train and we see the result obtained: what we expected was a quite clear split of the classes taken. After that we repeated the experiment with more classes passed and, as expected, the result was a bit worse than in the previous case: the division of the classes was not so clear and some items was not coded as well as before; this is a consequence of the amount of classes: more classes implicates a less precision in the classification.
The results obtained are shown in the following graphs:

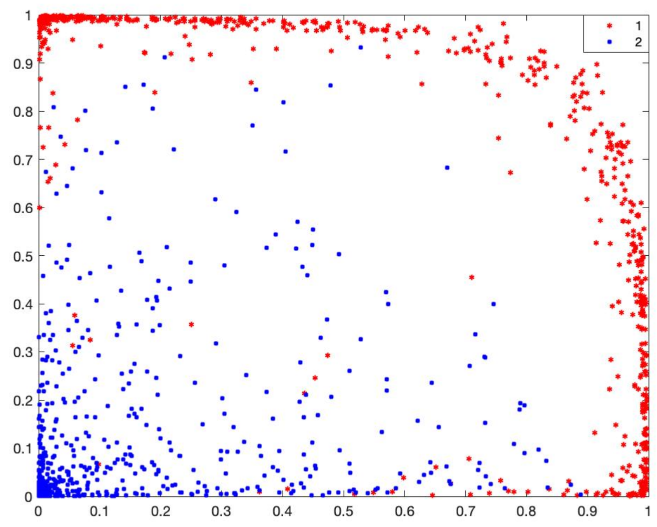


Figure 4.10: Graphical representation of the autoencoder output with class 1 and 2 as input.

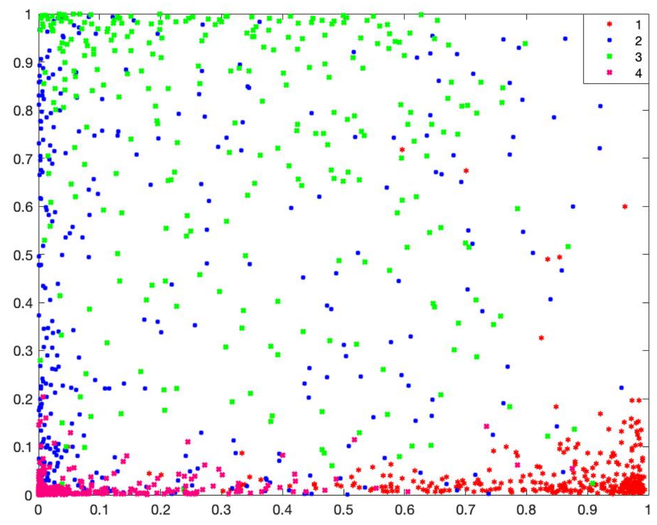


Figure 4.11: Graphical representation of the autoencoder output with class from 1 to 4 as input.

As it can be clearly seen from the images above, the first result takes only 2 classes and the division is better than the second one, processed with 4 different classes.
In particular many items of one specific class are decoded as they would belong to another one, thus introducing a minor accuracy and of course a bigger error in the classification.
All data are taken with the function $[X, T] = loadMNIST(0, [classesarray])$ in which the X is the matrix with 784 values for each class and the T is the matrix containing the labels for the class. All the figures can be shown by typing *imshow(reshape(X(row,:),28,28))*.

Moreover, there can be also some classifications with a small number of classes as input, two for example, that could be not so clear. The following figure makes it clearer:

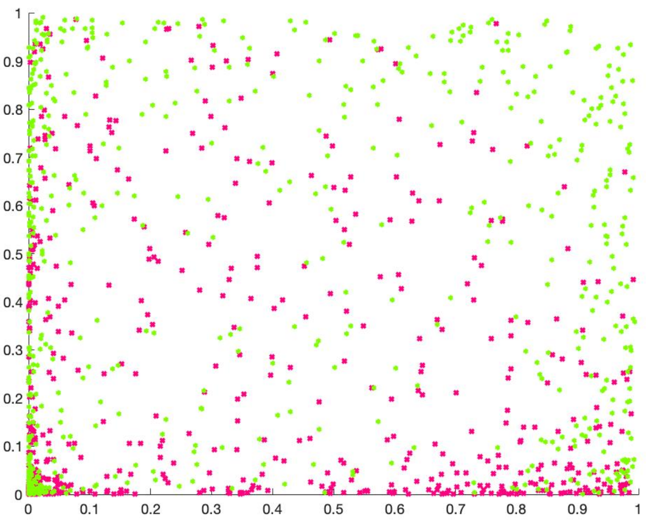


Figure 4.12: Graphical representation of the autoencoder output with class from 4 and 9 as input

As said before, this is a case in which, even if there are only 2 classes, their division is not so clear as expected. This can be a result due to some similarities in the classes and in the figures provided as data set that can be confusing during the classification: we can imagine that the number *4* and *9* could be confused if written not so clearly so this can bring lots of error, as in the figure.