Final Report Labs for the Machine Learning course

Nicholas Attolino

December 2023

Contents

Abstract

This document serves as a comprehensive overview of the labs within the Machine Learning course. The topics covered in both the labs and this report include:

- The Naive Bayes Classifier, which makes decisions based on a simplified assumption of attribute independence;
- Linear Regression, exploring the ability to create a linear model that represents the relationship between two or more variables;
- The k-Nearest Neighbors Classifier, which determines a class by measuring the distances between a given observation and the observations within the training set;
- Neural Networks, an interconnected system of artificial neurons utilized for diverse applications such as fitting data and classification purposes.

All the programs and the tests are implemented in MATLAB, a versatile platform that serves as a numerical computation and statistical analysis environment, as well as a programming language[7].
Each chapter is divided in:

1. Goal;
2. Data files overview;
3. Methods used;
4. Results.

1 Naive Bayes Classifier

1.1 Goal

The goals of this project are:

- Data pre-processing;
- Build a Naive Bayes Classifier;
- Improve the classifier with Laplace (additive) smoothing.

Bayesian classifiers assign the most likely class to a given example described by its feature vector.
Simplifying the learning process of these classifiers becomes easier by assuming that the features are unrelated to each other, then:

$$P(X|C) = \prod_{i=1}^{n} P(X_i|C) \tag{1}$$

where $X = (X_1, .., X_n)$ is a feature vector and C is a class.
Despite this unrealistic assumption, the resulting Naive Bayes classifier proves to be highly effective in real-world scenarios, frequently rivaling more complex methods[4].

1.2 Data Files overview

```
- Attolino-Lab1.zip
    - Attolino-Lab1
        - assignment1_ML.m
        - weather_data.txt
```

The *assignment1_ML* is the main program while *weather_data* is the dataset used.

1.3 Methods used

1.3.1 Dataset elaboration

The classifier will undergo testing using a Weather dataset.
Similar to any dataset, this one requires preprocessing before it can be utilized by automated programs.
Specifically, this dataset contains categorical features that must be transformed into numeric formats immediately upon loading the raw data into the MATLAB workspace.
Following these conversions, the entire data structure is altered, as the function employed to load the data produces a cell matrix that is challenging to handle.
This approach proves significantly faster compared to manual conversion, particularly with datasets containing hundreds of thousands of observations.
From this dataset, both the training and test sets are randomly extracted.
This task is accomplished using a MATLAB function that generates random numbers serving as indices.

1.3.2 Classification

Before the program starts the classification, it should check the given data are eligible to said classification.
First of all the test set's number of features should match the training set's one.
The number of columns could not match only when the training set has one more. This happens when the test set doesn't come with the target.
It's also important the single units of data are consistent. Since the data set is preprocessed converting every possible level to a specific integer, no one of them should be lesser than one. The program needs to know how many different classes it's going to classify the observations into.
To do that, it simply count the number of unique values in the target column of the training set. In order to compute the likelihood for each possible level of each attribute for every class, the software follows this algorithm:

Algorithm 1 Likelihood P(x=v|c) estimation

```
for each class C do
    compute the frequency of the class as number of occurrences
    - divided by the number of observations
    for each variable x do
        for each each possible value v for variable x do
            the number of instances of class c
            - that have variable x == value v
            - divided by the number of instances of class c
return likelihood
```

Practically, the returned likelihood matrix is a cell matrix with as many rows as the number of classes and as many columns as the number of attribute.
Each value of the matrix is an array that stores the frequency of each possible level of the specific attribute in the specific class. With this data, it's possible to compute the overall probability that a given observation belong to a class rather than another.

Algorithm 2 Discriminant function

```
for each observation X do
    for each class C do
        set the discriminant function of [X, C] equal to the frequency of C
        for each variable d do
            retrieve the frequency of the possible levels of d
            - when it belongs to C
            update the discriminant function of [X, C] multiplying it for
            - the frequency of the level of d in X, when the class is C
return discriminant function
```

Now it's possible to compare the values of the discriminant function for each observation. The probable belonging class is going to be the one with a greater value.
Once the program computes and stores the estimated target in a matrix, it's finally possible to elaborate the error rate. This action is obviously executed only if the test set target is given. It's equal to the number of incorrect classification divided by the number of observations.

1.3.3 Laplace Smoothing

Occasionally, especially when dealing with a small dataset like the one under consideration, certain combinations exclusively appear in the test set. In such instances, the program might attempt to calculate the frequency of a value that has never been encountered before.
Consequently, a specific statement might assign its probability as zero. While this approach prevents errors in the code, it is inaccurate. Absence of a particular value in the training set doesn't necessarily imply its probability is zero.
To rectify this behavior, the classifier function allows the option, when called, to employ the Laplace (Additive) Smoothing algorithm. This method presents an alternative approach for computing the probability of observing a specific value of a random variable x.
Knowing there have been N experiments and that the value i occurs n_i times, then:

$$P(x = i) = \frac{n_i}{N} \tag{2}$$

With Laplace Smoothing, the probability of observing value i is given by:

$$P(x = i) = \frac{n_i + a}{N + av} \tag{3}$$

where v is the number of values of the attribute x and a is a parameter that express how the data needs to be trusted from the program.
With a value of a greater than zero, it's possible to avoid the problem of zero probability. In order to implement this algorithm, a few changes to the code are necessary:

- First of all, the program needs to know the number of levels of each attribute. This information shall be given as an additional row of the training set;
- The program must remove the row with the number of levels before it starts to work on the training set. Also, if the program doesn't know at prior the number of levels for each attribute, it shall count the number of unique level from the training set;
- Ultimately, if the Laplace smoothing function is active, the updated way of computing the probability of observing a specific value needs to be used.

1.4 Results

As already mentioned, the data set used for valuating the classifier is the Weather data set. It has 14 instances, 4 attributes and 2 classes. It represents the decision problem of whether to go play outdoor or not, given a description of the weather.

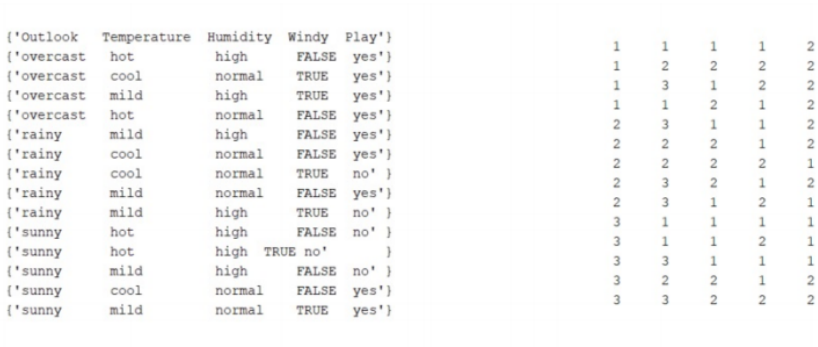

Figure 1: Weather dataset, before and after the pre-process phase

1.4.1 Standard Naive Bayes Classifier

The first test involves the standard configuration of the classifier (i.e. the one without the Laplace Smoothing function). There have been 100 calls to the classifier with different training and test set each time. The error rate ranges from 0 to 1 with an average of 0.44.

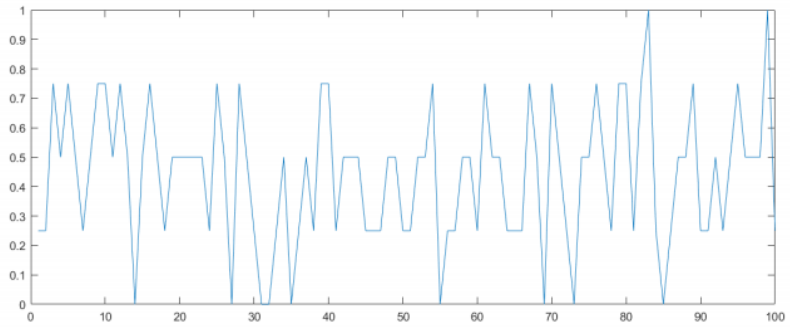

Figure 2: Error rate during the 100 episodes

1.4.2 Naive Bayes Classifier with Laplace Additive Smoothing

In order to see how the error rate changes with the variation of the a parameter, there have been 100 calls for each a in the range [0:0.1:2]. The minimum (best) average error rate was obtained for $a = 2$.
An a greater than 1 means that a prior belief of equally probable values is more trusted than the probability extracted from the data. The result then has its sense: the computed likelihood is based on a very poor training set, just 10 observations.

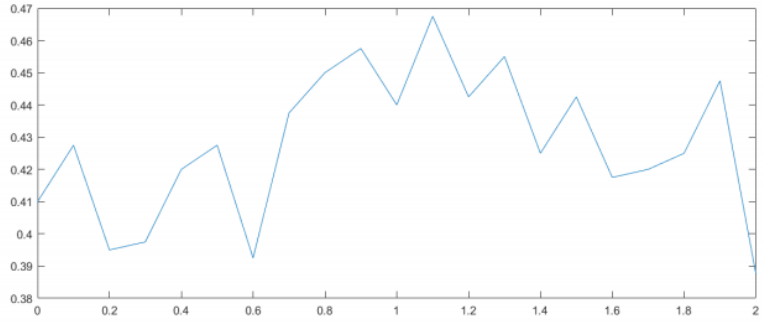

Figure 3: Error rate average with $a = [0 : 0.1 : 2]$

2 Linear Regression

2.1 Goal

The objective of this project is to create multiple Linear Regression models within the MATLAB environment.
What is a *Linear Regression*? The Linear Regression is a method that linearly models the connection between a singular output and one or more predictor variables, which are often referred to as the dependent and independent variables[5].
In this work, thehe models object focus on two datasets:

- turkish-se-SP500vsMSCI, a 536 by 2 matrix dataset;
- mtcarsdata-4features, a 32 by 5 matrix dataset.

After creating the models, multiple tests shall be executed.

2.2 Data Files overview

```
- Attolino-Lab2.zip
    - Attolino-Lab2
        - Linear_Regres_OneDim.m
        - Mean_Square_Error_OneDim.m
        - MotorTrends_Dataset.m
        - mtcarsdata.csv
        - turkish.csv
        - Turkish_Dataset.m
```

In the zip archive there are the files shown above, where we find:

- Linear regression function for one-dimensional problem;
- Mean Square Error function;
- Script of the tasks relatives to the Motor Trends Car dataset;
- Motor Trends Car dataset;
- Turkish dataset;
- Script of the tasks relatives to the Turkish dataset.

2.3 Methods used

2.3.1 Dataset elaboration

To load and make the datasets usable, the *readmatrix* function was used.
In particular, the Motor Trends Car dataset is split in 4 different column vectors to make it easier to work on and in this case the *Motor_Trends_Car_load* function has been made since it's often needed to repeat the division of the whole dataset.

2.3.2 One-dimensional case

Considering linear regression as an optimization problem, the goal is to find the parameters that minimize the mean value of loss over the entire dataset.
The mean value is the objective function and, choosing the mean square error as loss function, it looks like this:

$$J_{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - t_i)^2 \tag{4}$$

where N is the number of observations, t the target and y the estimated value. It's possible to find the minimum of the objective when its derivative is equal to zero.
When this happens, it's proved that:

$$w = \frac{\sum_{l=1}^{N} x_l t_l}{\sum_{l=1}^{N} x_l^2} \tag{5}$$

Thanks to this formula, the developed function called *Linear_Regres_OneDim* is able to fit a regression model: it receives in input a matrix of two columns, one for the observations and one for the targets, and returns the slope parameter.
To approximate the target, it is necessary to multiply the slope parameter with the observation x.
In particular, when this function receives the string *'withOffset'* as additional input, it will also return the offset parameter w_0:

$$w_0 = \bar{t} - w_1\bar{x} \tag{6}$$

where $\bar{x}$ and $\bar{t}$ are respectively the mean of x and t:

$$\bar{x} = \frac{1}{N}\sum_{l=1}^{N} x_l \qquad\qquad \bar{t} = \frac{1}{N}\sum_{l=1}^{N} t_l \tag{7}$$

In this case the slope have a different formula since it's computed by centering around the mean of the observations and the targets:

$$w_1 = \frac{\sum_{l=1}^{N}(x_l - \bar{x})(t_l - \bar{t})}{\sum_{l=1}^{N}(x_l - \bar{x})^2} \tag{8}$$

2.3.3 Multi-dimensional case

As for the multidimensional linear regression problem, it works differently: now there are d parameters and the data are composed of d-dimensional vectors so X is a matrix of N by d.
The goal becomes to make the model $\boldsymbol{y = Xw}$ as similar as possible to the

N-dimensional vector of the targets $\boldsymbol{t}$.
Accordingly, the minimum of the objective

$$J_{MSE} = \frac{1}{2}\|\mathbf{y} - \mathbf{t}\|^2 \tag{9}$$

is obtained by setting its gradient $\nabla J_{MSE} = 0$ and because of this, we will have:

$$w = (X^T X)^{-1} X^T \mathbf{t} \tag{10}$$

where $(X^T X)^{-1} X^T$ is the Moore-Penrose pseudoinverse of X, a formula that allow us to compute $\boldsymbol{w}$.
Finally we have:

$$\mathbf{y}^* = \mathbf{w}^* \cdot \mathbf{x}^* \tag{11}$$

where the value y^* estimates the most probable value of the output when a point $\boldsymbol{x^*}$ is received.

2.4 Results

2.4.1 One-dimensional problem without intercept for Turkish dataset

After invoking the linear regression function on the Turkish dataset, it yields the slope parameter as the output.
To visualize the way the function models the data, it's beneficial to create a plot comparing the first column of the dataset (observations) to the first column multiplied by the parameter w.

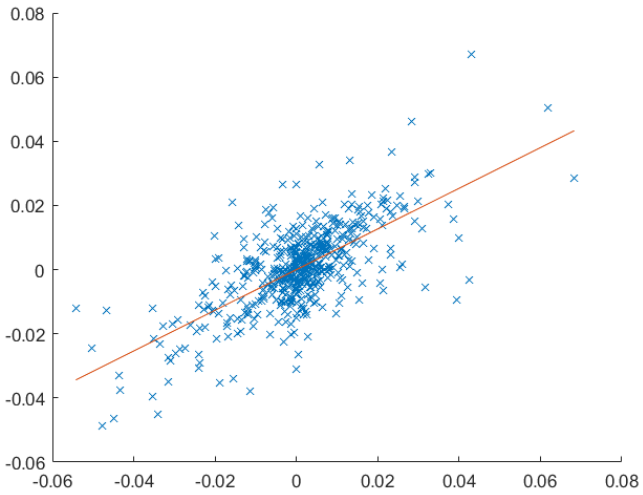

Figure 4: Scatter plot of the dataset with its least squares solution

The same test has been also carried out for ten times with random subsets of about 5% of the whole dataset showing this result:

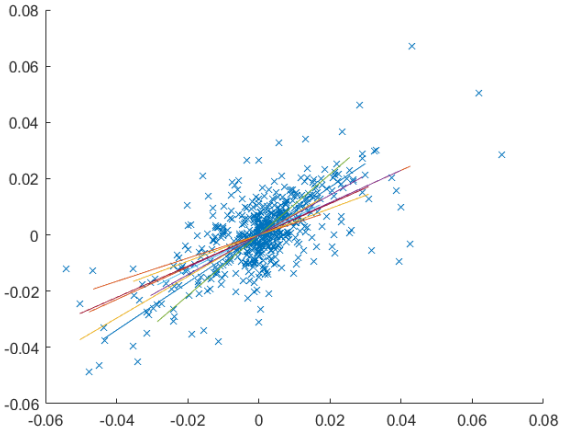

Figure 5: Scatter plot of the dataset with its least squares solutions

For each test, the software calculates the objective function for both the 5% data subset and the remaining 95% data subset.
As depicted in Figure 6, it's evident that the objective function value for the 5% subset is definitely lower than that for the 95% subset, demonstrating improved performance on the "training" data. This discrepancy arises from the model being specifically tailored to that dataset.

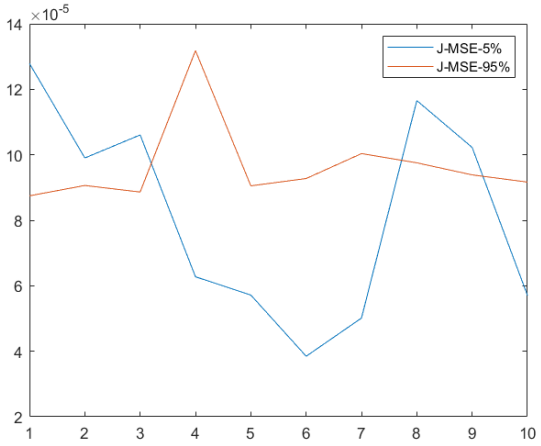

Figure 6: J_{MSE} over the 10 tests

2.4.2 Graphical comparison of solutions from two different Turkish subsets

The program provides users with the option to select between obtaining random data subsets from different ends of the Turkish dataset or from any point of it. While the preprocessing steps share similarities in both scenarios, they are not identical, necessitating the use of two distinct sets of commands.
In the case of obtaining random subsets from the different ends, the program first extracts the initial and final 20% of the complete dataset and subsequently selects half of the observations randomly from each of these subsets. This results in two subsets, each containing 10% of the total observations, drawn from the two extremes of the entire dataset.
Acquiring observations from any point of the dataset is a more straightforward process: the program generates n random indices and employs them to store the observations, where n corresponds to 10% of the total number of observations.

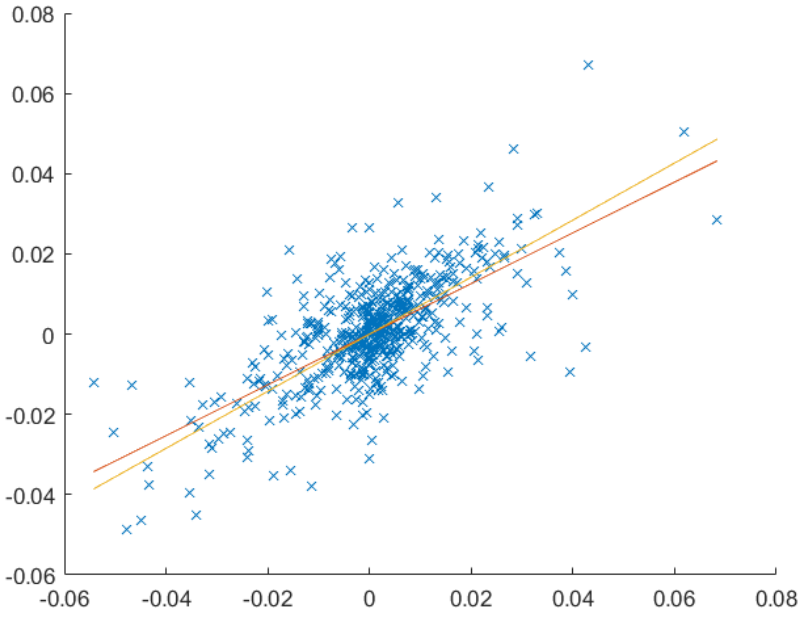

Figure 7: Comparison between solutions of different subsets

The point of taking data from different ends of the data set is that, since the data are collected across time, data collected in similar periods may be more similar than data collected from the beginning and the end of the whole period. In fact, in this case, the subsets' distribution are much different and so are their approximations.

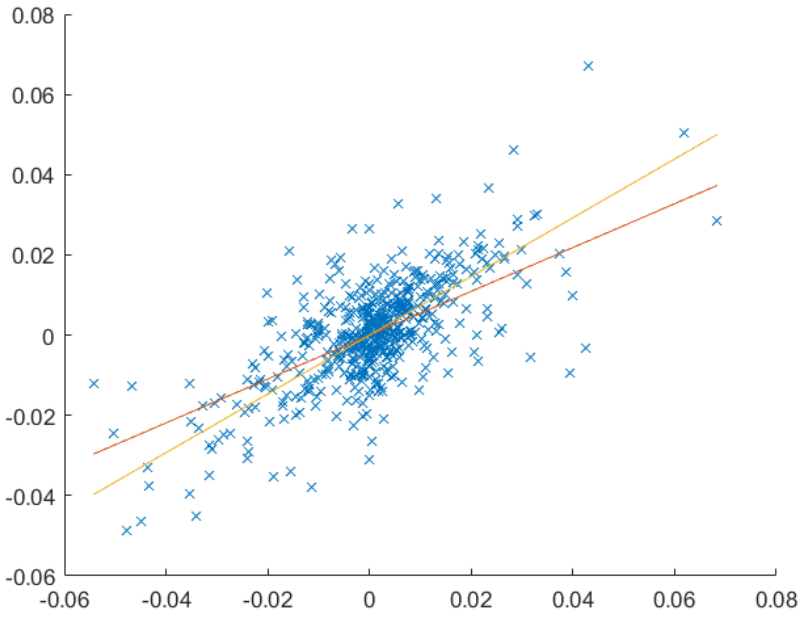

Figure 8: Comparison between solutions of different ends of the dataset

It's quite evident that the behavior remains fairly consistent when working with a random subset, resembling the behavior when the entire dataset is considered. This is primarily because the sampling algorithm selects instances that likely encompass the entire time span. These instances suffice to grasp the dataset's average behavior.

2.4.3 One-dimensional problem with intercept for Motor Trends Car

The program uses the weigh of a car to approximate its MPG (Miles Per Gallon). Figure 9 displays a scatter plot of these two attributes along with their linear approximation. In this case, the linear regression function also accepts the argument for requesting the intercept, which corresponds to the second output of the same function.

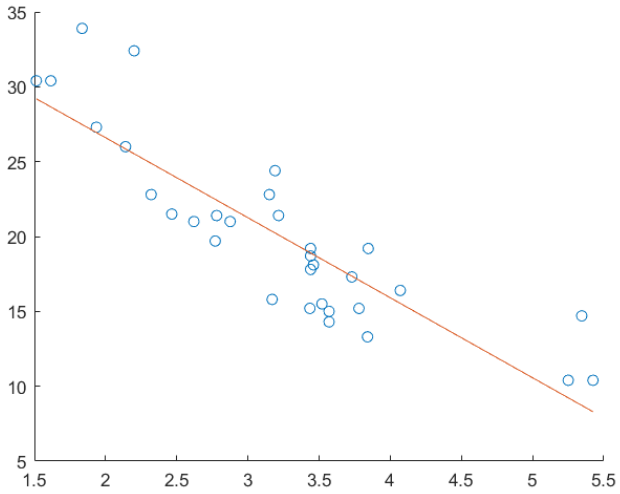

Figure 9: Weight vs MPG, linear approximation with intercept

An identical test was conducted ten times using random subsets, each comprising approximately 5% of the entire dataset. When fitting a model to only 5% of the 32 observations, it involves selecting just two data points and connecting them with a straight line:

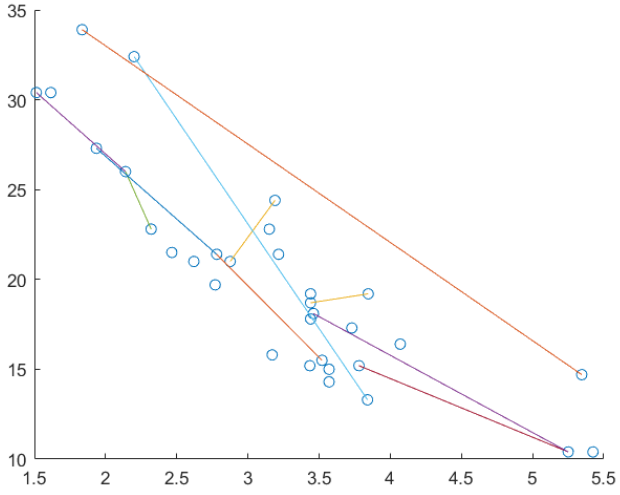

Figure 10: Scatter plot of the subsets with its least squares solutions

As there's only a single line connecting two points, the resulting slope parameter ensures that the algorithm will not miss, causing the objective to converge towards zero (on average, $J_{MSE} = 7.2575 \cdot 10^{-30}$).
When it comes to applying that parameter to the remaining observations, the algorithm will struggle to accurately estimate the relationship between the points (on average, $J_{MSE} = 0.8796 \cdot 10^{2}$).

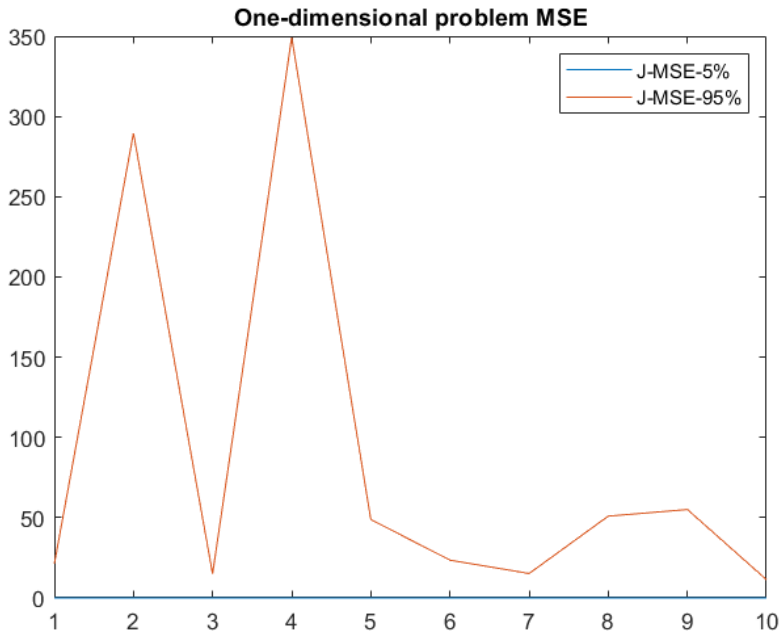

Figure 11: J_{MSE} over the 10 tests

This is the overfitting phenomenon: the creation of an analysis that matches a specific dataset too closely or perfectly, which can subsequently result in a failure to adapt to new data or make accurate predictions for future observations[8].

2.4.4 Multi-dimensional problem for Motor Trends Car

The program aims to discover a linear correlation between a car's Displacement, Horsepower, and Weight, and its Miles Per Gallon (MPG).
The provided solution managed to determine a slope parameter in this multidimensional scenario.
Just like in the one-dimensional problem, running the regression multiple times with an insufficient number of data points results in overfitting (on average, $J_{MSE} = 2.4526 \cdot 10^5$).
However, in this case as well, the objective for the 5% subset is higher than expected (on average, $J_{MSE} = 1.5907 \cdot 10^4$). This suggests that the relationship between the analyzed parameters is far from being linear.

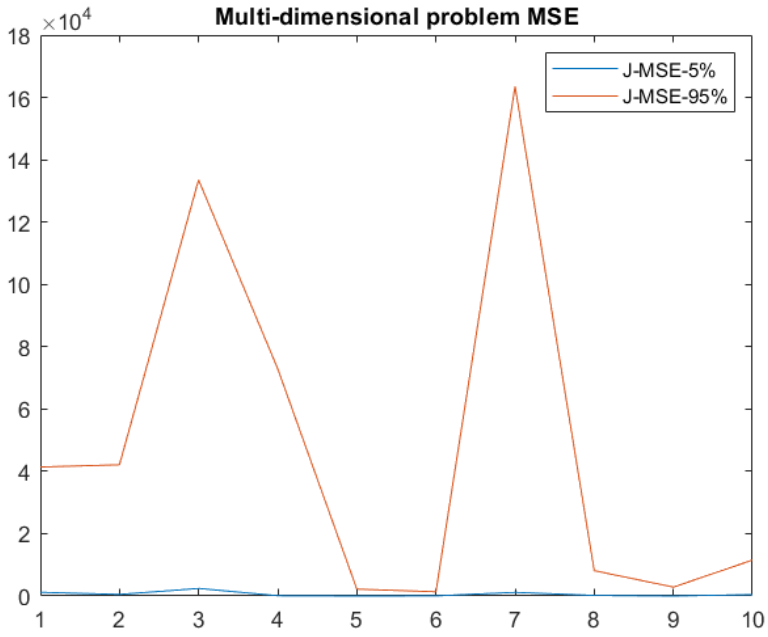

Figure 12: J_{MSE} over the 10 tests

3 k-Nearest Neighbors Classifier

3.1 Goal

The goals of this project are:

- Obtain a dataset;
- Build a kNN classifier;
- Test the kNN classifier.

The k-nearest neighbors algorithm, abbreviated as k-NN, functions as a non-parametric method for classification. Instead of creating a model with defined parameters, it constructs a discriminatory rule directly from the available data. The dataset employed for both training and testing is known as the *mnist* dataset. This dataset contains images of handwritten digits depicted in grayscale, each measuring 28 by 28 pixels. These images are divided into 10 distinct classes corresponding to the numbers ranging from 0 to 9.
In the training set, each class comprises a minimum of 5800 instances, while in the test set, there are at least 890 instances for each class.

3.2 Data Files overview

```
- Attolino-Lab3.zip
    - Attolino-Lab3
        - Lab3.m
        - mnist
            - (the dataset files)..
```

The *Lab3* is the main program while *mnist* is the dataset used.

3.3 Methods used

3.3.1 Dataset elaboration

Due to the abundance of instances within each class in the dataset, there's a need to downsize the training set. This involves selecting a defined percentage from the entire dataset.
The data set contains classes arranged in sequence. To ensure equal representation of all classes within the subset, it's imperative to randomly select observations.
An alternate approach involves generating individual subsets for each class and subsequently merging these 10 subsets to compose the training set.

3.3.2 kNN Classifier

The Nearest Neighbor rule is based on the assumption that observations which are close together (in some appropriate metric) will have the same classification[1].
So, given a training set:

$$X = \{x_1, ..., x_l, ..., x_n\}$$

and a query point $\bar{x}$, the estimated class is the same as the point:

$$q = arg\ min||x_l - \bar{x}||$$

The k-Nearest-Neighbour is a variant of the previous algorithm: it takes from the training set the firsts k observations ordered by crescent euclidean distance

$$\{n_1, ..., n_k\} = top\ k||x_l - \bar{x}||$$

and chose as class the most frequent between them

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

3.3.3 The Matlab Function

The classifier function has at least 4 input arguments:

- The **training set**, a *n x d* matrix where n is the number of observations and d is the number of attributes. In the studied case each observation has 784 (28x28) attributes: each image, sampled as grey scale, is represented as a row vector of 784 numbers;
- The **training set labels**, a *n x 1* matrix with each value equal to the number the corresponding observation represents. Note that the "0" class is represented by the number 10, in order to use it as index;
- The **test set**, a *m x d* matrix where m is the number of observations to classify and d remain the number of attribute. One of the firsts operations the function does, is to check if the d-s are the same;
- The **k** number of neighbors. This could be either a scalar or a vector of k-s. The function will return as many estimated classes as the number of k-s in the vector.

Eventually, there is an additional input that might be the **test set classes**. When the latter is present, it is used to compute the accuracy once the function classify all the observations.

The program executes a number of checks before it starts the classification:

1. Initially it checks that the number of arguments are at least 4;
2. After that, it checks that the number of attributes of training and test sets matches;
3. Finally it checks that all the values in the k vector are greater than 0 but smaller than the total number of observations (The function cannot find 10 nearest neighbors if there exist only 5 observations).

The main operation the function does is to call another function, the *pdist2* function [9]: this returns the euclidean distance between the two given observations (if we don't specificate it, by default it returns always the euclidean distance).
When two matrices (training and test sets) are given in input, it returns a matrix with the distances of each observation in the first matrix from each observation in the second one.

```
>> X = rand(2,2)
Y = rand(2,2)
D = pdist2(X,Y,'euclidean')

X =

    0.7922    0.6557
    0.9595    0.0357


Y =

    0.8491    0.6787
    0.9340    0.7577


D =

    0.0614    0.1747
    0.6524    0.7225
```

Figure 13: The *pdist2* function

An optional input argument it has is the '*Smallest*' string followed by an integer n: for each observation in the test matrix, *pdist2* finds the n smallest distances by computing and comparing the distance values to all the observations in the training matrix.
The function then sorts the distances of every test observation in ascending order; in this case, the second output contains the indexes of the observations in the training set corresponding to the distances just computed.

```
>> [D,I] = pdist2(X,Y,'euclidean','Smallest',2)

D =

    0.0614    0.1747
    0.6524    0.7225


I =

     1     1
     2     2
```

Figure 14: The *pdist2* function with a second input/output

The function set up provides an 'n' that matches the largest 'k' value in the corresponding vector, given that additional neighbors aren't necessary.
With every 'k' value and for every test observation, the classifier calculates the class label by identifying the mode among the initial 'k' class labels obtained using the earlier indices.
The result will be a matrix having rows equivalent to the number of test set observations and columns equivalent to the quantity of values in the 'k' vector. For each observation in the test set, the function delivers the most likely class for each 'k'.

3.4 Results

The test involved utilizing the entire training set, along with 50% of the test set, while employing specific k values:

$$k = [1, 2, 3, 4, 5, 10, 15, 20, 30, 40, 50]$$

Using a *for* loop, the function was invoked to categorize each test set observation into either the i_{th} class or any other.
As the test set labels were provided as input, the function became capable of determining the accuracy in correctly identifying each digit against the remaining digits, considering various k values within the specified vector.

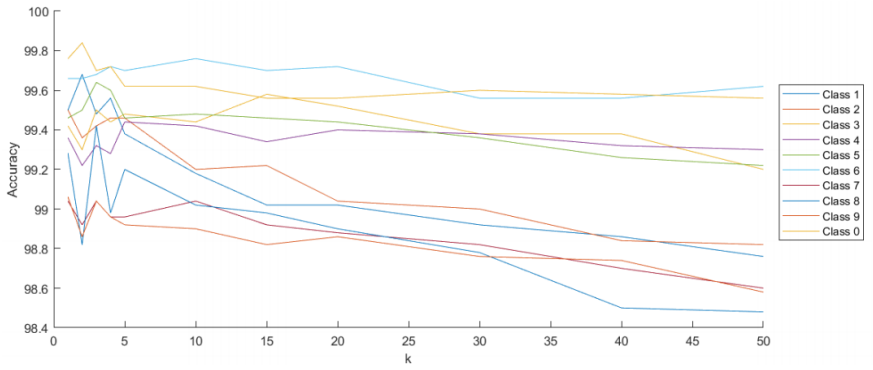

Figure 15: Class Accuracy variation with k

With $k = 3$, the accuracy spanned between 99.04 and 99.7, averaging at approximately 99.42, making it the highest average accuracy. The better accuracy was achieved when identifying the digit 0 using $k = 2$, reaching around 99.84.

4 Neural Networks

4.1 Goal

The requests of this project are:

- Neural networks in Matlab;
- Feedforward multi-layer networks (multi-layer perceptrons)
- Autoencoder.

The objective of this project is to get acquainted with the MATLAB Neural Network Toolbox trying to classify some patterns from chosen datasets and ultimately to train a multi-layer perceptron as an autoencoder for the *MNIST* dataset.
The Deep Learning Toolbox offers powerful tools like the following:

- The **Neural Network Fitting App**, a system that, given a dataset with a very simple configuration, will train a shallow network to fit the given data;
- The **Neural Network Pattern Recognition App** that can train a shallow neural network to classify patterns;
- The **Neural Network Clustering app**;
- The **Neural Network Time series app**.

In this project the firsts 2 tools were used.

4.2 Data Files overview

```
- Attolino-Lab4.zip
    - Attolino-Lab4
        - Task1
            - iris_data.txt
            - Task1.m
            - wine_data.txt
        - Task2
            - class_pair_plots
                - ...
            - mnist
                - ...
            - plotcl.m
            - Task2.m
        - Task0.m
```

In the zip archive there are the files shown above, where we find:

- Task0 MATLAB code;
- Task1 MATLAB code;
- iris_data is the first dataset for Task1;
- wine_data is the second dataset for Task1;
- Task2 MATLAB code;
- plotcl is the function to plot data;
- mnist is the dataset folder for Task2;
- class_pair_plots is a folder with some encoded data plotted.

4.3 Methods used

4.3.1 Classify Patterns with a Neural Network

Thanks to the Neural Network Pattern Recognition tool, it's possible to train a shallow network with custom datasets.
The tool can either be used through a graphical interface or by command line. This work rely on the Iris[2] dataset and on the Wine[3] dataset: the *Iris* dataset has 150 instances, 4 numeric attributes and 3 classes; while the *Wine* dataset is a 3-class dataset with at least 48 occurrences per class and 13 continuous attributes.

4.3.2 Autoencoders

Training an autoencoder involves training a multi-layer perceptron neural network where the desired target aligns with the provided input.
Consequently, the network assimilates a compressed representation of the data, enabling the hidden layer to recognize and categorize passively certain patterns. The data object of this study are hand-written digits in a 28 by 28 grey-scale matrix, while the autoencoder needs 784 input and output units and the hidden layer has 2 units.

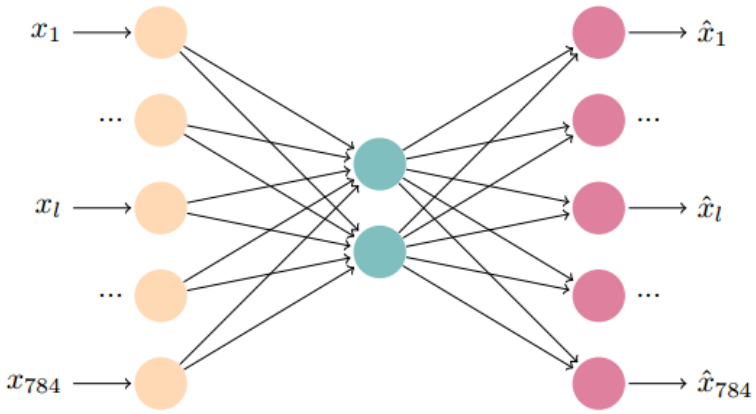

Figure 16: Autoencoder Neural Network

4.3.3 The Matlab Program

The developed program conducts training for an autoencoder using a training set that consists of only 2 categories of handwritten digits. Subsequently, it encodes the same dataset through the trained autoencoder. Plotting the encoded data will show the pattern recognition ability of the network.
Both the training set and test sets undergo processing via a custom function named *data_pre_processing*. This function requires input specifying the two desired categories and the percentage of available observations to utilize. It generates a matrix containing the specified observations and arranges the matrix in a random order.
To plot the encoded data is used the *plotcl* function: this takes the encoded observations and their classes and plots them with a color that depends on the class they belong.
It's crucial to highlight that the autoencoder remains unaware of the labels assigned to the observations throughout its operation. These labels serve solely for the graphic feedback provided by the *plotcl* function since distinguishing the classes allows understanding if the autoencoder has been able to recognize any patterns.

4.4 Results

4.4.1 Pattern classification

The developed program empowers the user to select which one of them shall be used to train the network.
Following this choice, a series of minor preprocessing instructions are carried out. The mentioned tool requires two matrices: the predictors matrix contains the essential input data crucial for predictions, while the responses matrix holds the corresponding classes for each observation.
After several simple operations, both matrices will be ready. For the network to function effectively, it necessitates information on the desired training function among the available options, the dimensions of the hidden layer, as well as the sizes allocated for the training, validation, and test sets.
Once these parameters are established, the program proceeds to execute the training and testing phases.
The training process persists until a specific stopping condition is fulfilled. In these instances, the training persists until there is a consecutive rise in validation error for six iterations (*Met validation criterion*). When this happens it means the network is probably overfitting.
The tool itself offers multiple plots to understand the results of the test, like the Confusion matrices in Figure 17 and the ROC Curves in Figure 18.

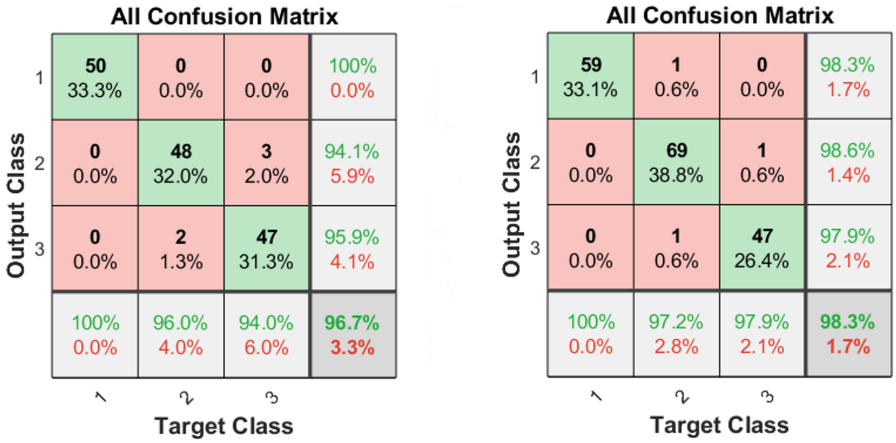

Figure 17: Confusion matrices for the Iris and the Wine datasets respectively

The accuracy of the network's outputs can be observed by examining the number of correct classifications, depicted in the green (diagonal) squares. These squares indicate instances where the output class aligns with the target class. On the other hand, the red squares highlight chases where the network incorrectly assigned observations to a different class.

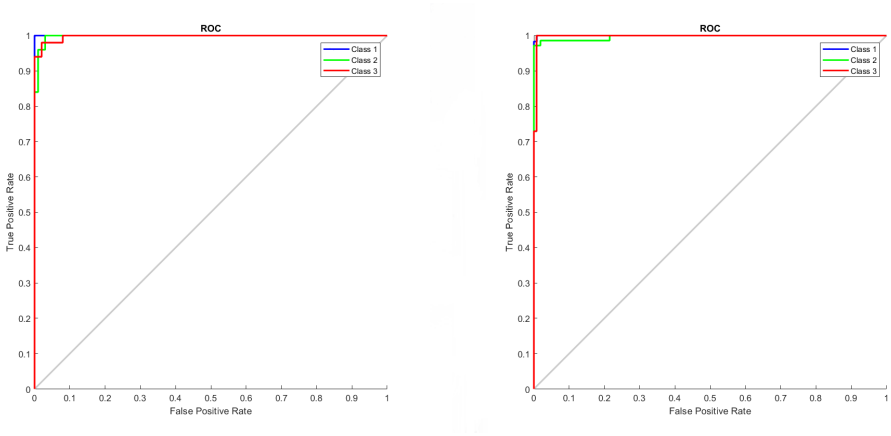

Figure 18: ROC Curves for the Iris and the Wine datasets respectively

The colored lines 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 these problems, the networks performs very well.

4.4.2 The trained autoencoder

The test has been carried out on 7 pairs of classes. The function loops on a matrix with the 7 pairs as rows, and executes the training and the encoding.

$$Classes = \begin{bmatrix} 1 & 8 \\ 10 & 8 \\ 10 & 1 \\ 2 & 6 \\ 2 & 9 \\ 4 & 1 \\ 1 & 6 \end{bmatrix}$$

For the training, the set had about 60% of the available data, meanwhile for the test, only a 5% of data was used.

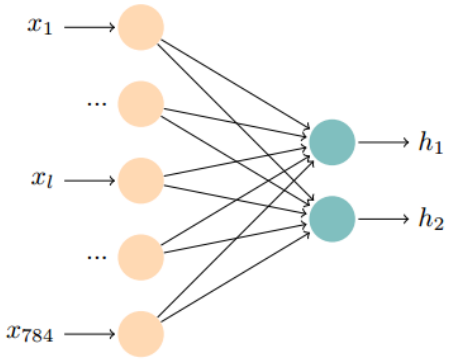

Figure 19: Encoder

By plotting the encoded data of every pair of classes, a scatter plot in two dimensions shows the encoder's capability for recognizing patterns.

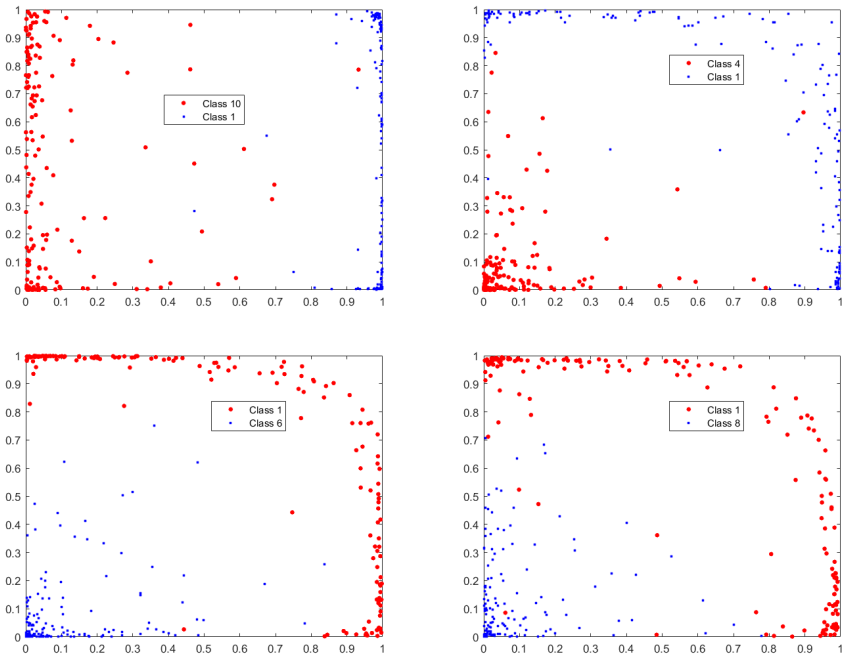

Figure 20: Plots of encoded data with some almost clear decision regions

For some digits the work is tougher and though there are not well shaped regions:

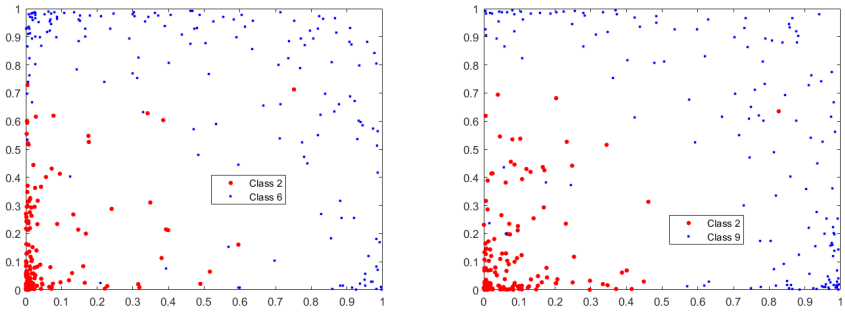

Figure 21: Plots of encoded data with blurry decision regions

It's noteworthy that these outcomes were achieved without the network having prior knowledge of the specific class to which each individual observation belongs.
This represents an instance of unsupervised learning, where the autoencoder autonomously identified certain patterns by attempting to replicate the input as the output.
This occurred subsequent to a compression phase caused by a hidden layer containing fewer units than the input/output layer.

5 Conclusions

Naive Bayes Classifier

Assuming independence among features, the Naive Bayes classifier represents an optimal model, wherein no other classifier is anticipated to attain a lower misclassification error rate[6].
However, when this assumption is not true and practically in the vast majority of real-world classification problems, the naive Bayes classifier has demonstrated excellent performance.

Linear Regression

As anticipated, a linear association between two variables can be quantified within the parameters of a model, typically yielding a minuscule margin of error. However, challenges arise when dealing with sparsely populated datasets or when no linear correlation exists among certain attributes. In the former scenario, overfitting is likely to occur, while in the latter, the outcome is a considerably larger objective value and subpar model performance.
Furthermore, a visual illustration was presented, showcasing how randomly selected data points can effectively capture the overall behavior of the entire dataset.

k-Nearest Neighbors Classifier

The classifier's performance varies based on the specific class it aims to differentiate from the rest.
Overall, the classifier demonstrates high effectiveness, achieving an accuracy surpassing 98.8% when using the appropriate k value.
Directly providing a K vector to the classifier, rather than a single scalar each time, significantly enhances the execution speed by circumventing unnecessary repetitions in the computation process.

Neural Networks

The MATLAB Deep Learning Toolbox demonstrated its immense utility in this context.
It enables swift and straightforward training of neural networks, serving both fitting and pattern recognition objectives (those are the object of this work). Moreover, the provided visualizations aid in assessing a trained neural network. The autoencoder is a concrete proof of how the neural networks can distinguish different classes of data without a prior clue about their differences.
A possible next step involves using stacked autoencoders[10] in order to have a network with multiple hidden layer trained one by one.

References

[1] T. Cover and P. Hart, *Nearest neighbor pattern classification*. 1967, vol. 13, pp. 21–27. DOI: 10.1109/TIT.1967.1053964.

[2] R. A. Fisher, *Iris*. 1988, DOI: https://doi.org/10.24432/C56C76.

[3] S. Aeberhard and M. Forina, *Wine*. 1991, DOI: https://doi.org/10.24432/C5PC7J.

[4] I. Rish, *An Empirical Study of the Naïve Bayes Classifier*. Jan. 2001, vol. 3.

[5] D. Freedman, *Statistical Models : Theory and Practice*. Cambridge University Press, 2005, ISBN: 0521854830.

[6] D. Berrar, *Bayes' Theorem and Naive Bayes Classifier*. Jan. 2018, ISBN: 9780128096338. DOI: 10.1016/B978-0-12-809633-8.20473-1.

[7] T. M. Inc., *MATLAB Version: 23.2.0.2365128 (R2023b)*. The MathWorks Inc., 2023.

[8] IBM, *What is overfitting?* [Online]. Available: https://www.ibm.com/topics/overfitting.

[9] *pdist2 function*. [Online]. Available: https://it.mathworks.com/help/stats/pdist2.html.

[10] *Stacked Autoencoders*. [Online]. Available: https://it.mathworks.com/help/deeplearning/ug/train-stacked-autoencoders-for-image-classification.html.