Mattia Sarti

Machine learning techniques for real-time collision detection in a wheeled mobile robot

Master's thesis in Reliability, Availability, Maintainability and Safety
Supervisor: Shen Yin
Co-supervisor: Lippi Marco, Xingheng Liu
October 2023

Norwegian University of Science and Technology
Faculty of Management Engineering
Department of Mechanical and Industrial Engineering

NTNU

ABSTRACT

This thesis focuses on the application of various machine learning (ML) techniques to improve the overall dependability of a wheeled mobile robot (WMR) control during a real-time simulation.
The whole system consists on a WMR, provided by the company 'Quanser', and a related ground station that represents a Digital virtual twin of the device, which aims to control the real-time simulation of the robot.
The control model of the WMR had already been developed in Simulink, where it was possible to immediately carry out various experiments in order to gain some insights into possible improvements to make the whole system safer and more reliable, which is the aim of this work.

This was done by first extracting a large dataset containing all the signals that occurred during the simulation. After analyzing the data, different ML models were developed in Python that were able to identify collisions and their duration. At the end of this step, the ML model with the highest score was selected and, as a result, a neural network model was identified as the best one.
The neural network was then exported to the robot's Simulink Digital twin, allowing the model's performance to be verified in real-time during the simulation.

The main objective of this work was to prevent the robot from malfunctioning and to enable the bumper to react appropriately even in such situations.
Since it makes sense that the sensors would lose their ability to work properly after a certain period of time, the ML models developed aim to work in a real-time simulation in parallel with the standard model, which includes the output of the sensors, in order to make the device more resistant to sudden failures and able to act as if these failures were not occurring.
In industrial applications, the robot's sensors may stop working as they should, causing downtime and potential hazards to nearby workers.
In this way, it was possible to make the robot's bumper sensors more reliable and safer, behaving as if they were not affected by these faults, making the system more flexible and durable.

The final results clearly show how the application of ML algorithms in the field of robot control can lead to significant improvements in safety and reliability.

CONTENTS

ABBREVIATIONS

List of all abbreviations in alphabetic order:

- **ANN** Artificial neural network
- **DT** Decision Tree
- **KNN** K-nearest neighbors
- **LR** Logistic Regression
- **ML** Machine learning
- **NB** Naive Bayes
- **SVM** Support vector machine
- **WMR** Wheeled mobile robot
- **XNN** Explainable neural network

CHAPTER

ONE

INTRODUCTION

1.1 Hardware information

Contemporary research is placing a significant emphasis on mobile robots due to their potential utility across a spectrum of challenging environments. These environments include agriculture and harvesting, household chores, and medical applications. The fundamental technologies underpinning these applications encompass sensory-enhanced remote control of mobile robots, the ability for self-localized autonomous navigation, obstacle avoidance, and intelligent decision-making for task execution, [1].
The effective use of these methods relies heavily on a range of sensors, including rotary encoders to determine the robot's position, range finders to help avoid obstacles and vision systems to recognise visual patterns [2].
Investigations into mobile robots and their sensor technologies have become an essential component of automation research.
The evolution of control algorithms for mobile robots necessitates the availability of a well-suited robotics platform furnished with the requisite sensors.
To address this challenge, the company "Quanser" has introduced a mobile robot control framework that uses MATLAB Simulink interactive graphical environment alongside a versatile collection of customizable block libraries, [3]. This work involves the kinematic and dynamic control of the WMR called "Qbot 2e".

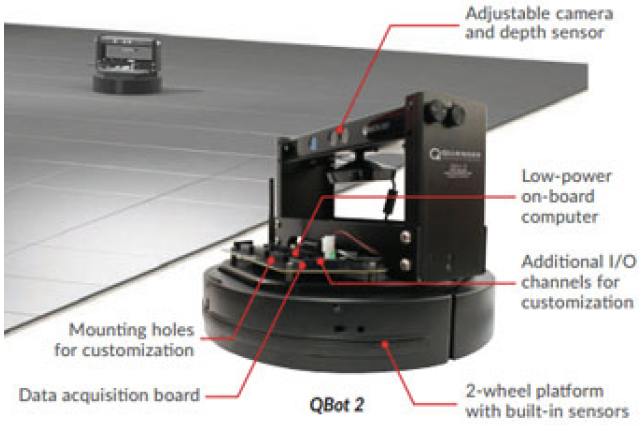


Figure 1.1.1: Visualization of Qbot 2e

The work is based on a system consisting of the WMR "Qbot 2e" and an associated accurate digital virtual twin of it that behaves in the same way as the physical hardware and can be measured and controlled using MATLAB/Simulink and other development environments.
In Table 1.1.1 it is possible to see the details about the device that we worked with, [4].

Characteristic	**Value**
Diameter	35 cm
Height (with Kinect mounted)	27 cm
Maximum linear speed	0.7 m/s
Available payload	4.5 kg
Battery life	Maximum 3 hours
On-board computer	Raspberry Pi™ with integrated WiFi
Camera resolution	640 x 480
Depth sensing	11 bit
Depth sensor range	0.5 - 6 m
On-board sensors	3 digital bump sensors 2 wheel encoders 2 digital wheel drop sensors 3 digital buttons 3 cliff sensors 2 over current sensors 1 3-axis gyroscope 1 battery voltage sensor 2 analog motor current sensors 1 Kinect RGBD sensor 1 Z-axis angle measurement (heading) 1 charger 2 multi-color programmable LEDs
Additional I/O channels available	28 reconfigurable digital I/O channels, including: 1 SPI bus channel 1 I2C serial bus channel 2 PWM output channels 1 UART serial port (interface 3.3 V serial device)

Table 1.1.1: QBot 2e specifications

1.2 Motivation

In recent trends, Artificial Intelligence (AI) is used for the creation of complex automated control systems. Learning ability in robotics has recently experienced significant hurdles as a result of the theoretical advancement at the boundary between optimization and ML, even though the latter has long been recognized as a core technique in the many domains of robotics. In fact, it has been shown that combining ML and optimization can significantly improve decision-making quality and learning capacity in decision systems, [5].
The development of autonomous mobile systems, in particular WMRs, has recently accel-

erated due to their widespread use in many industries. These device are used in situations where operational safety and efficiency are critical, such as warehouse logistics and autonomous cars, [6].
Traditional collision detection systems often use static sensor thresholds and predefined rules, which limits their ability to react to changing circumstances. On the other hand, the use of ML algorithms offers a promising way to improve collision detection and prediction capabilities.
Additionally, the successful implementation of ML-driven collision methods in WMRs not only enhances safety but also contributes to their overall autonomy and efficiency, [7]. This research addresses a critical need in the field of robotics and autonomous systems, where real-world applications demand sophisticated, data-driven solutions for ensuring the seamless operation of WMRs.
Consequently, the primary motivation for this work lies in the potential to exploit the power of ML in order to create intelligent and adaptive collision detection systems for WMRs.
By retrieving dataset consisting of signals that occurred during many simulations and then training ML models on these dataset, we aim to develop algorithms that can work in parallel with the default model which are capable of learning complex patterns and making real-time decisions to increase the reliability of the system, making it more flexible, durable and cost-saving.

1.3 Project Description

The core problem we are addressing is the improvement of real-time bumper control for WMRs, with a particular focus on scenarios where traditional sensor-based approaches may fall short. We aim to develop solutions that enable the robot to respond effectively in the face of sensor failures that cause poor performance.
Our research objectives include the development and implementation of ML algorithms that aims to predict the exact moment of collision and the right duration of it during the real-time simulation.
In addition, we aim to create a system that allows for a seamless transition between sensor-based control and ML-based control, making the WMR more resilient and able to adapt over time.
The work began with the extraction of the dataset, including all the signals coming from the robot during each simulation. Later the dataset was used to build a ML models on Python, with different methods in order to obtain the best score from the final model prediction based on the test set.
Since the best model turned out to be an Artificial neural network, (ANN), the goal of this step was to transfer the trained ANN model to the Qbot Digital twin Simulink model, so that the ML model could also perform in real-time, which is the purpose of this work.
This made it possible to create an algorithm that works in parallel with the standard Qbot model, with the aim of replicating the same behaviour of the bumper sensors throughout the simulation.
Finally, it was necessary to modify the Qbot Simulink model to make the ML model suitable for this software, adding various blocks to shape and modify the standard Digital

twin model.
Lastly we will discuss different logic and methods used to achieve our purpose and further potential improvements on this scenario.

CHAPTER

TWO

THEORY

2.1 Motion models of Qbot 2e

2.1.1 Kinematic model

The kinematic controller is a Lyapunov based nonlinear feedback control, where the controllers are implemented on the system hardware using QUARC, a MATLAB-Simulink based software, [8]. The kinematic control model has a forward velocity v_c and the angular velocity ω_c as the control inputs and the error in position as the output. The goal of the kinematic controller is to drive the error to zero.

$$v_c = \frac{\nu_L + \nu_R}{2} \tag{2.1}$$

$$\omega_c = \dot{\theta} = \frac{\nu_R - \nu_L}{d} \tag{2.2}$$

Where d is the distance between the left and right wheels, θ is the heading angle of the robot, ν_L is the left wheel velocity, and ν_R is the right wheel velocity. The purpose now is to explain the motion of a mobile robot with a two-wheel differential drive in a Cartesian frame. It is necessary to build two coordinate systems in order to explain how the mobile robot will travel. The first one is a global coordinate frame, where the origin is a point in a two-dimensional plane; the second is a local coordinate frame, where the origin is the centroid of the robot.

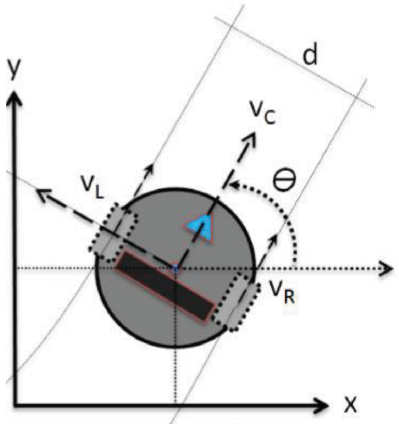


Figure 2.1.1: Quanser QBot 2e Mobile Platform Reference frame definition, [8]

The robot's posture q is defined by the following position vector.

$$q = [x\ y\ \theta]^T \tag{2.3}$$

By taking the derivative of posture and mapping the robot's linear and angular velocities, ν_C and ν_C, to its cartesian frame, the kinematic model can be stated as follows:

$$\dot{q} = \begin{bmatrix} \dot{\chi} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v_c \\ \omega_c \end{bmatrix}$$

From equation previous above, it is then reasonable to conclude that the trajectory can be controlled by adjusting the linear velocity and angular velocity of the Qbot, [8]. The use of a local frame of reference is implicit in the construction of the kinematics model discussed above. In other words, rather of using the global frame that would be utilized in an environment map, the chassis speed, ν_C, is stated in the forward/backward (heading) direction of the robot chassis. since the robot chassis heading changes when the angular rate is non-zero, $\omega_C = \dot{\theta}$, it is necessary to apply a transformation to the differential drive kinematics model in order to compute the robot chassis motion with respect to the global reference frame. The needed transformation for a robot with a heading of is the rotation matrix shown below:

$$R = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.4}$$

This transformation maps motion expressed with respect to the robot chassis local frame to the corresponding motion in the global frame, [9]. Defining state vector, S, as the position, x and y, and the heading, θ, of the robot chassis. Its definition and rate of change are given as follows:

$$S = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix}, \quad \dot{S} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} \tag{2.5}$$

The x and y axis lie in the ground plane where the robot primarily travels in. The heading θ is measured about the vertical z axis, which is defined as positive pointing upwards. The heading is zero, $(\theta = 0)$, when the robot chassis' forward direction aligns with the global x axis. The rate of change of the states can be expressed in terms of the robot chassis speed, ν_C , and angular rate, ω_C as follows, [9]:

$$\dot{S} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = R \begin{bmatrix} v_C \\ 0 \\ \omega_C \end{bmatrix} = \begin{bmatrix} v_C \cos\theta \\ v_C \sin\theta \\ \omega_C \end{bmatrix} = \begin{bmatrix} \frac{1}{2}(v_R + v_L)\cos\theta \\ \frac{1}{2}(v_R + v_L)\sin\theta \\ \frac{1}{d}(v_R - v_L) \end{bmatrix} \tag{2.6}$$

Eventually, equation 2.6 represents the forward kinematics model for the QBot 2e that computes the linear speed, $\dot{x}$ and $\dot{y}$, and turning rate, ω_C, of the robot chassis given its heading, θ, and wheel speed, ν_R and ν_L.

Regarding the Dynamic control, it provides a more comprehensive representation of a robot's behavior, it takes into account also the external forces and toques applied to the robot and then describes how it will respond to various forces and moments applied to it.

In comparison to the kinematic model, the dynamic model is more difficult to formulate and solve since it necessitates a thorough comprehension of the physical characteristics and rules of motion of the robot. It does, however, give a more accurate picture of how the robot would perform in dynamic circumstances and while interacting with the environment. It is essential for applications where precise control, high-speed motion, or interactions with the environment are critical, such as this device.
In summary, the kinematic model simplifies the robot's behavior by focusing on motion and position, while the dynamic model provides a more complete representation, considering forces and torques.

2.1.2 Dead reckoning model

Dead reckoning is the procedure for obtaining the location based on the previous known position, [10].
It refers to a method of estimating the current or future position of a robot or vehicle based on its previously known position and measurements of its speed and direction of travel.
It is used in robotic applications in order to reduce the need for sensing technology, such as some specific sensors, GPS, or the placement of some linear and rotary encoders in an autonomous robot, [11]. Dead reckoning is then a navigation method based on measurements of distance traveled from a known point used to incrementally update the robot pose. This leads to a relative positioning method, which is simple, cheap and easy to accomplish in real-time, [12].
WMR frequently estimate their position from *odometric* data, which is the use of data from motion sensors to estimate change in position over time.
The Qbot has rotatory encoders, which is is an electro-mechanical device that converts the angular position or motion of an axis into analogue or digital output signals, [13]. Odometric data is then used by a robot from its encoders to retrieve some estimations.
The results of these measurements were then used for the robot's odometric localization where the pose at time instant t in the global reference frame was obtained by integrating equation 2.6 over the interval from 0 to t, [14]:

$$\begin{bmatrix} x(t) \\ y(t) \\ \theta(t) \end{bmatrix} = \int_0^t \begin{bmatrix} \frac{1}{2}(v_R + v_L)\cos\theta \\ \frac{1}{2}(v_R + v_L)\sin\theta \\ \frac{d}{2}(v_R - v_L) \end{bmatrix} dt \tag{2.7}$$

This pose will then be used to control the robot in such a way that it moves along the planned path.

2.2 Bias and Variance

In order to understand the goodness of a ML model it is important to understand prediction errors, *bias* and *variance*.
The term "bias" refers "any basis for choosing one generalization hypothesis over another, other than strict consistency with the observed training instances, [15]".
According to [16], in statistics the term "bias" is the persistent or systematic error that the learning algorithm is expected to make on training sets, when is trained.
Building a ML model there is always a trade off problem between the model's ability to

minimize bias and variance at the same time, which is related to the concepts of underfitting and overfitting.
Since a ML model try to approximate the exact formula that describes the relationship among the predictors, roughly speaking bias and variance could been also seen as the inability to capture the true relationship and the fitting difference over data between training and test set, respectively.
Models with high variance focus a lot on training data and does not generalize on the test set which it has not seen before. As a result, these models perform very well on training set but has high prediction error on predicting over the test set.
Then, *bias* and *variance* are defined as follows:

$$Bias(\hat{y}) = E[\hat{y}] - y \tag{2.8}$$

$$Variance(\hat{y}) = E[(E[\hat{y}] - \hat{y})^2] \tag{2.9}$$

It is possible to see the limit of a ML model mathematically by [17]:

$$(y - \hat{y})^2 = (y - E[\hat{y}] + E[\hat{y}] - \hat{y}) \tag{2.10}$$

Applying the expectation operator on both sides, the left term becomes the Mean Square Error (MSE) while the right-hand side of the equation develops as follows:

$$\begin{aligned} MSE &= E[(y - \hat{y})^2] && (2.11)\\ &= E[(y - E[\hat{y}] + E[\hat{y}] - \hat{y})^2] && (2.12)\\ &= E[(y - E[\hat{y}])^2] + E[(E[\hat{y}] - \hat{y})^2] + 2E[(y - E[\hat{y}])(E[\hat{y}] - \hat{y})] && (2.13)\\ &= E[(y - E[\hat{y}])^2] + E[(E[\hat{y}] - \hat{y})^2] + 2E([y - E[\hat{y}])(E[\hat{y}] - E[\hat{y}]) && (2.14)\\ &= (y - E[\hat{y}])^2 + E[(E[\hat{y}] - \hat{y})^2] + 0 && (2.15) \end{aligned}$$

Since the expected value of a constant is simply the constant itself and $E[-E[\hat{y}]] = -E[\hat{y}]$ it is possible to simplify the first term.
Instead, the third term goes to zero.

$$\begin{aligned} MSE &= (y - E[\hat{y}])^2 + E[(E[\hat{y}] - \hat{y})^2] + 0 && (2.16)\\ MSE &= Bias^2 + Variance + \epsilon && (2.17) \end{aligned}$$

Where:

- y represents the actual data point.
- $\hat{y}$ represents the predicted data point after having fitted the model.
- ϵ measures the irreducible error that comes from every model prediction (included in $\hat{y}$), which is the error that can't be reduced by creating good models. It is a measure of the amount of noise in the data.

Consequently the purpose is to build a ML model that aims to minimize error that will be present, due to the noise of the data and the limits of the model.
Minimizing the Total Error means that the complexity of the model should be stopped after a while during the fitting, over the training set.
The model complexity is defined as a measure of how accurately a ML model can predict unseen data, as well as how much data the model needs to see in order to make good

predictions.
Then model complexity is important because it determines how generalizable a model is, meaning how well the model can be used to make predictions on new, unseen data, (i.e. on the test set).

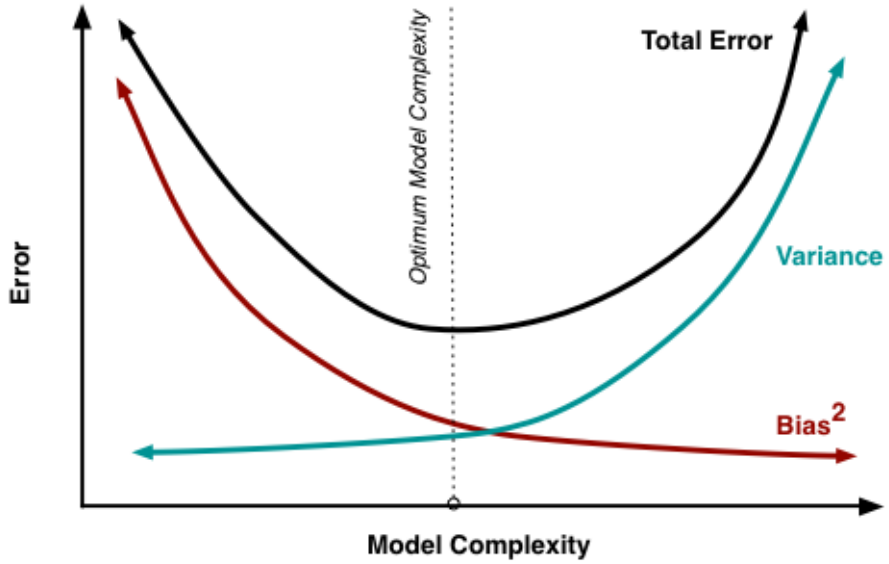


Figure 2.2.1: Total error curve that shows its minimum as a balance between Bias and Variance, which is the point where the model should be stopped to train
The region at the left of the exact Model Complexity (dotted line) is called 'Underfitted' while the region at the right is known as 'Overfitted'.

Without falling in Overfitting, there are different techniques that could be used in order to obtain a good learning for the model, such as:

- Use **early stopping**: Early stopping is a technique that is adapt to prevent overfitting. It involves training the model until the validation error starts to increase and then stopping the training process. This ensures that the model does not continue to fit the training data after it has started to overfit.

- Use **k-fold cross-validation**: Cross-validation is a technique that can be used to reduce overfitting by splitting the data into multiple sets and training on each set in turn.
  This approach involves randomly dividing the set of observations into k groups, or folds, of approximately equal size.
  While the kth fold is treated as a validation set, the model is trained on the remaining $k-1$ folds. This process is applied k times every time with a different k fold as a validation set.
  This allows the model to be trained each time on different data and prevents it from being overfitted to a particular set of data.
  The k results of a k-fold cross-validation are often summarized with the mean of the model skill scores.

- Monitor the performance of the model as it is trained and adjust the parameters accordingly.

2.3 Decision tree

A Decision tree (DT), is a supervised ML algorithm that is used for both classification and regression tasks.
The mechanism behind decision trees is that of a recursive classification procedure as a function of explanatory variables and supervised by the target variable. Moreover, this mechanism sees a recursive splitting of the initial sample alongside one variable at the time into two or more subsamples, [18].
The main idea is to start by splitting the variable which is more able to explain the response, setting it as the root of the tree. Then, in the same way, it continues by splitting recursively each subsample, called node, into smaller nodes alongside single variables and according to threshold values that identify two or more branches.
Finally, when a node is no longer split into further nodes (either because a stopping criterion is reached or because it is no longer useful for the evaluation), the actual node becomes a *leaf* of the tree, which represents the class of the sample or a value, if it is a regression problem.
The leaf may hold also a probability vector indicating the probability of the target.
In the simplest and most frequent case, each test considers a single attribute, such that the instance space is partitioned according to the attribute's value. In the case of numeric attributes, the condition refers to a range.

Weather	**Temperature**	**Humidity**	**Wind**	**Play**
Sunny	Hot	High	Weak	No
Cloudy	Hot	High	Weak	Yes
Sunny	Mild	Normal	Strong	Yes
Cloudy	Mild	High	Strong	Yes
Rainy	Mild	High	Strong	No
Rainy	Cold	Normal	Strong	No
Rainy	Mild	High	Strong	Yes

Table 2.3.1: Table that represents the first rows of the dataset

In Table 2.3.1, the first four columns are called *features* and the last column is called *target* or *response variable*. In any ML model, features variables are used to predict the target variable.

Instances are classified by navigating them from the root of the tree down to a leaf, according to the outcome of the tests along the path.

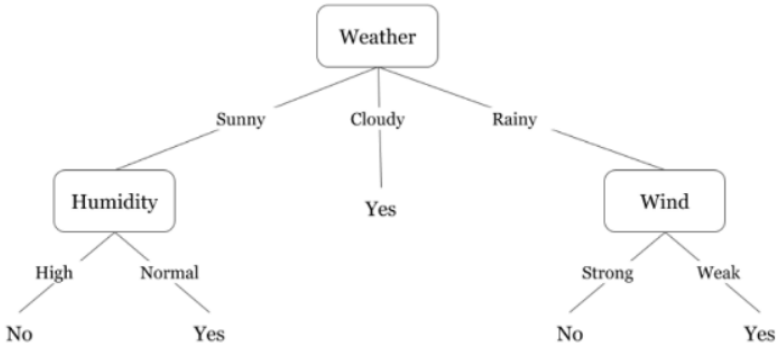


Figure 2.3.1: Representation of decision tree

The Figure 2.3.1 describes a DT that reasons weather a person should play tennis or not based on the previous table. As it possible to see, a single sample is classified by simply navigating from the root node to the leaf.
The tree complexity, which has a crucial effect on its accuracy, is explicitly controlled by the stopping criteria used and the pruning method employed, [19].
Usually the tree complexity is measured by these of the following metrics:

- Total number of nodes
- Total number of leaves
- Tree depth
- Number of attributes used

2.3.1 Algorithmic framework for Decision Trees

Classification tree
In order to check "the goodness of splitting criterion" for evaluating how well the splitting is, various splitting indices were proposed, where the difference among them is the definition of the importance of all the variables in explaining the response.
First of all it is important to define the *entropy*, that is the degree of uncertainty, impurity or disorder of a variable and characterizes the impurity of an arbitrary class (target variable).
So if all elements belong to a single class, then it is termed as "Pure", and if not then the distribution is named as "Impure", with a certain degree of impurity.
The formula of Entropy is given by:

$$H = -\left(\sum_{i=1}^{n} p_i \log_2 p_i\right) \tag{2.18}$$

Where:
H is the Entropy
p_i is the proportion of the class i in the dataset
n is the total number of the classes present on the target variable

With more than one attribute taking part in the decision-making process, it is necessary to decide the relevance and importance of each of the attributes.

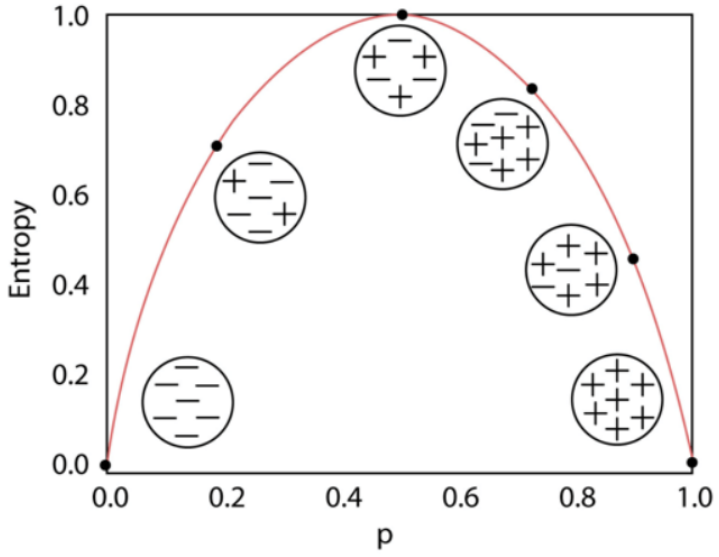


Figure 2.3.2: Representation of entropy in a binary classification problem, [20]

Thus, placing the most relevant feature at the root node and further traversing down by splitting the nodes. As we move further down the tree, the level of impurity or uncertainty decreases, thus leading to a better classification or best split at every node.
Subsequently it possible to define some of different splitting criteria such as:

1. Information Gain
2. Information Gain Ratio
3. Gini Index

2.3.2 Information Gain

Information gain (IG), is one of the measures used to select the best attribute at each step in growing the tree.
It is computed for each variable in the dataset where the variable that has the largest IG is selected to split the dataset. Generally, a larger value of IG indicates a smaller entropy. IG is calculated in this way:

$$IG(S, a) = H(S) - H(S|a) \tag{2.19}$$

Where, [21]:
$IG(S, a)$ is the information for the dataset S given by a random variable a.
$H(S)$ is the entropy for the dataset S before any change.
$H(S|a)$ is conditional entropy for the dataset after choosing a random variable a for splitting it into two nodes.

The IG could be seen also as the difference of the Entropy H in the dataset S before and after the splitting of the dataset.

Note that minimizing the entropy is equivalent to maximizing the information gain.[22]
The process of selecting a new attribute (variable) and partitioning the training examples

is now repeated for each non terminal descendant node, this time using only the training examples associated with that node.
Attributes that have been incorporated higher in the tree are excluded, so that any given attribute can appear at most once along any path through the tree.
In most decision tree implementations, it is possible to specify a 'max depth', that is the number of value as a hyperparameter to limit the maximum depth of the tree. In this case, the maximum depth is limited to the specified value, which may be less than the number of attributes in the dataset.
In summary, the maximum depth of a decision tree can be at most equal to the number of attributes in the dataset, but in practice it will generally be less than this value in order to avoid overfitting and achieve a more generalised model.

Finally, IG tends to favor attributes with a large number of values because they can potentially create many homogeneous subsets, leading to deeper and more complex trees. According to the IG, when an attribute has many distinct values, it can lead to better divisions in the data, resulting in more specific rules and potentially overfitting the training data.
Then the Information Gain Ratio (IGR) was introduced to provide a correction for this bias towards attributes with many values.
It normalizes the Information Gain by taking into account the intrinsic characteristics of the attribute, such as the number of distinct values it can take.
By doing so, it reduces the impact of attributes with many values, making the attribute selection process more balanced.
IGR is computed as follows:

$$IGR(S,a) = \frac{IG(S,a)}{H(S|a)} \tag{2.20}$$

It has been shown that the IGR tends to outperform simple IG criteria, both from the accuracy aspect, as well as from classifier complexity aspects, [23].

2.3.3 Gini Index

Gini Index is a powerful measure of the randomness or the impurity in the values of a dataset, it aims to decrease the impurities from the root nodes to the leaf nodes of a decision tree model.
Consequently Gini index G(S), over a dataset, represents the probability of classifying a data point imperfectly.

$$G(S) = \sum_{i=1}^{K} P(i) * (1 - P(i)) \tag{2.21}$$

Where:
$P(i)$ is the probability of picking a data point of class i, that is the proportion of examples which belongs to that class.
$1 - P(i)$ is the probability of not picking a data point of class i.
K is the number of classes.

The Gini Index varies between 0 and 1, where 0 represents purity of the classification where all the data points belong to one class and then there is a perfect classification, on

the other hand 1 denotes a random distribution of points among various classes.
A Gini Index between 0 and 1 indicates a certain degree of impurity.

Regression tree
In this scenario where the purpose is no longer assign labels to new data points, the goal now is building a regression model where each leaf represents a numeric value, [18].
Here the tree stratifies or segments the predictor space into different non-overlapping regions, known as leaves of the tree or terminal nodes.
It is possible to summarize the process in two steps:

1. Dividing the predictor space, that is the set of possible values for $X_1, X_2, ..., X_p$ into J distinct and non-overlapping regions, $R_1, R_2, ..., R_J$.

2. Every observation that falls into the region R_j, is characterized to have the same prediction, which is simply the mean of the response values for the training observations in R_j.

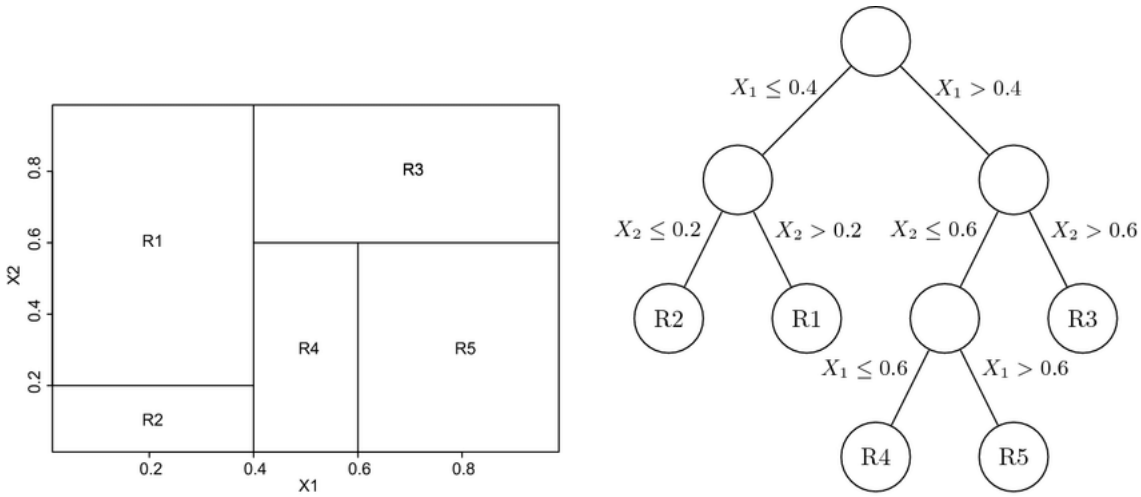


Figure 2.3.3: A partition of two-dimensional feature space with different thresholds values that results in the corresponding tree

According to [18], the process to build an optimal regression tree consists in searching different threshold t within the predictor space $X = X_1, ..., X_p$ where the latter that has the smallest value of RSS becomes a candidate for becoming the discriminant in order to split the actual node. With the need of searching the boxes $R_1, ..., R_J$ the goal is to minimize the RSS (Residual Squares Error), given by:

$$RSS = \sum_{j=1}^{J} \sum_{i \in R_i} (y_i - \hat{y}_{(R_j)})^2 \tag{2.22}$$

Where:
y_i represent the actual target point i
$\hat{y}_{(R_j)}$ is the prediction that represents the mean response for the training observations within the jth box. $R_1, R_2, ..., R_J$ represent the leaves of the tree.

Subsequently the predictor space X_p is divided into J distinct and non-overlapping regions, $R_1, R_2, ..., R_J$, where every observation that falls into the region R_j , has the same prediction, which is simply the mean of the response values for the training observations in R_j.
Finally the process ends when one of the stopping criterion is reached, such as the number of observations included in each box or leaf.

Since this problem is classified as *NP-hard*, it is computationally infeasible to consider every possible partition of the feature space into J boxes for every step, for this reason it is better to proceed with a greedy algorithm, which is a top down approach called *recursive binary splitting.*
The *recursive binary splitting* approach is top-down because it begins at the top of the tree (where at every point all observations belong to a single region) and then successively splits the predictor space, [18].
Each split is indicated via two new branches further down on the tree. It is greedy because at each step of the tree-building process, the best split is made at that particular step, rather than looking ahead and picking a split that will lead to a better tree in some future step.

Recursive binary splitting:

1. Consider all predictors $X_1, ..., X_p$, and all possible values of the cutpoints t for each of the predictors.

   REPEAT:

2. For any j and t it possible to define a pair of half-planes such that:

$$R_1(j,t) = \{X \mid X_j < t\} \qquad R_2(j,t) = \{X \mid X_j \geq t\} \tag{2.23}$$

3. Cut the predictor space X with threshold t and predictor X_j searching the value of j and t that minimize the equation below:

$$\sum_{i:x_i \in R_1(j,t)} (y_i - \hat{y}_{R_1})^2 + \sum_{i:x_i \in R_2(j,t)} (y_i - \hat{y}_{R_2})^2 \tag{2.24}$$

4. Split one of the two previously identified regions with the previous process.

   UNTIL(Stopping criterion is reached)

5. Found all the features regions $R_1, ..., R_J$, for a given test observation predicts the value using the mean of the training observations in the region to which that test observation belongs.

2.3.4 Pre-pruning and post-pruning

The process described about how to build a DT may produce good predictions on the training set, but is likely to overfit the data, leading to poor test set performance. This is because the resulting tree might be too complex. A smaller tree with fewer splits (that is, fewer regions $R_1, ..., R_J$) might lead to lower variance and better interpretation at the cost of a little bias.
There are 2 possible ways to prevent Overfitting:

1. **Pre-pruning**: the process of decision-making for every node is made during each stage of the splitting the tree where the cross-validation error is monitored. If the value of the error does not decrease significantly enough then the growth of the decision tree should be stopped.

2. **Post-pruning**: growing a very large tree T_0, and then prune it back in order to obtain a subtree which produce better performance on the test set.

Cost complexity pruning
However post pruning on decision trees is considered more mathematically rigorous, then one post-pruning strategy that is possible to use is called *Cost complexity pruning*, also known as *weakest link pruning*.
After having built a beginning full tree T_0 with the mechanism described previously, now instead of considering every possible subtree in order to prune some nodes, consider a sequence of trees indexed by a non-negative tuning parameter α.
For each value of α chosen within a sufficiently large list of candidates, it is possible to compute the $TreeScore$ which identifies a sequence of best subtrees, [18]:

$$TreeScore = \sum_{m=1}^{|T|} \sum_{x_i \in R_m} (y_i - \hat{y}_{R_m})^2 + \alpha |T| \qquad \alpha = 1, ..., L \tag{2.25}$$

Where:

- $|T|$ indicates the number of terminal nodes of the tree T.
- R_m is the subset of predictor space corresponding to the mth terminal node.
- $\hat{y}_{R_m}$ is the mean predicted response associated with R_m.
- α acts as penalty, since controls a trade-of between the subtree's complexity and its fit to the training data.
  Different values of α give a sequence of trees, from full sized tree (T_0 with α=0), to a tree with one leaf.
  As α increases, there is a price to pay for having a tree with many terminal nodes, on the other hand as α decreases allow the tree to have more complex and deeper branches.

Found the scores for each tree the next step is to use *k-fold cross validation* where different values of α are tested on multiple subsets of data.
Finally the value of α that minimize the average error, $\alpha^\star$, is the optimal value to take in order to build the final decision tree which has the best performance to generalize the data (low bias and low variance).

2.4 Logistic Regression

The purpose of Logistic Regression (LR), is to determine the probability of the occurrence of an event by exploiting many of the principles of linear regression, logistics can provide accurate information on the probability of the occurrence of events [18].
The LR model, instead of modeling the response variable Y directly, as it would do the Linear Regression model, aims to compute the probability that Y belongs to a particular category.
In order to build a trustworthy model, it requires certain prior assumptions to be met. Satisfying these assumptions is essential to ensure that the regression results are not affected by bias.

- Lack of strongly influential outliers that could influence the data
- Lack of correlation between the features, i.e. multicollinearity
- Independence of the observations
- Linearity in the logit function, i.e. sigmoid function, for continuous variables
- Large enough sample to avoid overfitting

An important difference concerning these assumptions is that LR does not make many assumptions which are, on the other hand, essential to the Linear regression, such as:

- Linear relationship between the dependent and independent variables
- Residuals (error terms) does not need to be normally distributed
- Homoscedasticity is not required, that is present when the size of the error term is different across values of an independent variable

As regards to the last point, [24], since the Linear Regression model use Ordinary least squares (OLS) as a technique for estimating the coefficients, the outcome gives equal weight to all observations, but when there is a violation of homoscedasticity the cases with larger disturbances could have a greater influence to the final result than other observations.

2.4.1 LR Model

As already mentioned LR models the probability of the response variable, using for convenience the generic 0/1 coding for it, where it is possible to express the conditional probability:

$$P(X) = P(Y = 1|X) \tag{2.26}$$

Since the purpose now is to model the relationship between the above equation, would not be consistent to adopt a straight line to fit a binary response that is coded as 0 or 1, the consequence is that in this case predict P(X) $<$ 0 for some values of X and P(X) $>$ 1 for others. To avoid this problem, it is better to model P(X) using a function that gives an output between 0 and 1 for all values of X.

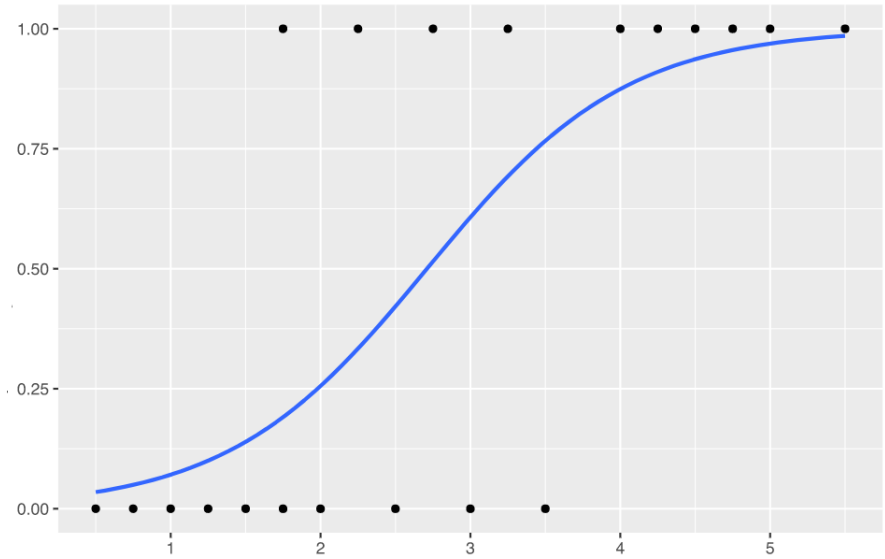


Figure 2.4.1: Logistic function that returns a probability to classify a new data point

Many functions meet this description. For reproducing the shape of a S-curve, in logistic regression it is common to use the logistic function, known also as a sigmoid function:

$$p(X) = \frac{e^{\beta_0+\beta_1 X}}{1 + e^{\beta_0+\beta_1 X}} \tag{2.27}$$

To fit this model there is a method called *maximum likelihood*, which aims to compute the best coefficients in order to give to the new data points the best logistic function resulting from the method.
In this scenario, the y-axis is confined to probability values between 0 and 1 and since during the fitting it is preferable to have a list of values in order to have a meaningful outcome, then it's more convenient to do a bit of manipulation to the previous equation, that becomes:

$$\frac{p(X)}{1 - p(X)} = e^{\beta_0+\beta_1 X} \tag{2.28}$$

The quantity $\frac{p(X)}{1-p(X)}$ is called *odds*, and can be any value between 0 and ∞. Always keeping in mind that the assumption started with the equation 2.28, if the value is close to 0 indicate very low probabilities of the response variable otherwise a value close to ∞ means an high probability of the outcome.
By taking the natural logarithm of both sides of (3.18), resulting in:

$$\log(\frac{p(X)}{1 - p(X)}) = \beta_0 + \beta_1 X \tag{2.29}$$

The left term is called *logit* and the 2.29 equation is helpful because the logit transformation produces a linear function of the parameters $\beta_0, ..., \beta_n$, (where the extended formula is: $\beta_0 + \beta_1 X_1 + ...\beta_n Xn$), in addition the range of values *logit* is between $-\infty$ and $+\infty$ which makes it more appropriate for fitting the model.

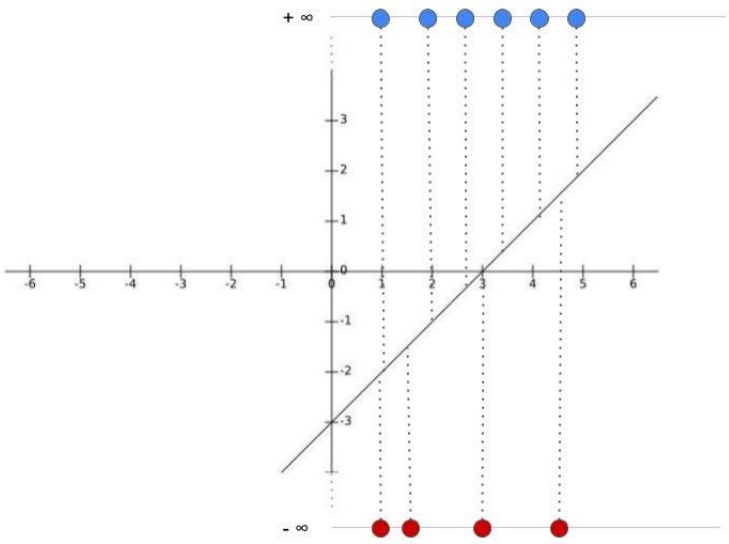


Figure 2.4.2: Transformation of the y-axis from the probability to log(*odds*)

Since the fitting process needs to project the data points onto a straight line in order to compute a log(*odd*) value, the picture above 2.4.2 shows the y-axis transformation, which pushes the data points to $+\infty$ and -∞, in such a way as to find the best fitting line for the LR model, which eventually will correspond to the best S-curve.

According to [25], the process to choose the best logistic function and then to find the best parameters starts from defining the *likelihood* function:

$$p(y_1, ..., y_n) = \prod_{i=1}^{n} p(X_i)^{y_i}(1 - p(X_i))^{1-y_i} \tag{2.30}$$

Now, since $\frac{p(X_i)}{1-p(X_i)} = e^{(\beta_0+\beta_1 X_i)}$ and $1 - p(X_i) = \frac{1}{1+e^{(\beta_0+\beta_1 X_i)}}$ the equation 2.30 can be expressed as:

$$p(y_1, ..., y_n) = L(\beta_0, \beta_1) = \prod_{i=1}^{n} e^{(\beta_0+\beta_1 X_i)^{y_i}}(\frac{1}{1 + e^{(\beta_0+\beta_1 X_i)}}) \tag{2.31}$$

Keeping in mind that $p(X_i) = \frac{e^{(\beta_0+\beta_1 X_i)}}{1+e^{(\beta_0+\beta_1 X_i)}}$ and computing the derivative of log-likelihood function with respect to β_0 and β_1 respectively, gives the following equations:

$$\frac{\partial l(\beta_0, \beta_1)}{\partial \beta_0} = \sum_{i=1}^{n} y_i - \frac{e^{(\beta_0+\beta_1 X_i)}}{1 + e^{(\beta_0+\beta_1 X_i)}} = 0 \tag{2.32}$$

$$\frac{\partial l(\beta_0, \beta_1)}{\partial \beta_0} = \sum_{i=1}^{n} (y_i - p(X_i)) = 0 \tag{2.33}$$

$$\frac{\partial l(\beta_0, \beta_1)}{\partial \beta_1} = \sum_{i=1}^{n} y_i X_i - X_i \frac{e^{(\beta_0+\beta_1 X_i)}}{1 + e^{(\beta_0+\beta_1 X_i)}} = 0 \tag{2.34}$$

$$\frac{\partial l(\beta_0, \beta_1)}{\partial \beta_1} = \sum_{i=1}^{n} X_i(y_i - p(X_i)) = 0 \tag{2.35}$$

Finally, starting from the equations 2.33 and 2.35 it possible to compute the optimal parameters. Since finding solutions to these equations is analytically difficult, then it is preferable to use the numerical iteration method called Newton Raphson, which will not be treated in this work but it can be found in the article [26].

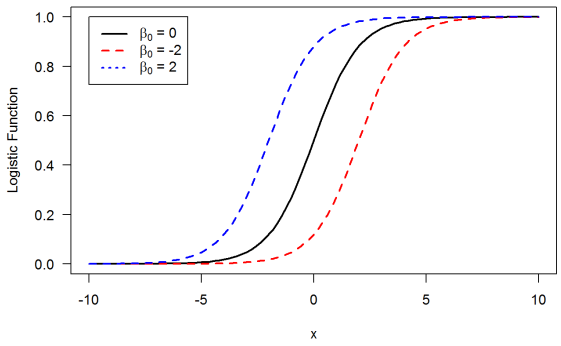


Figure 2.4.3: Different values of $beta_0$ in the logistic function

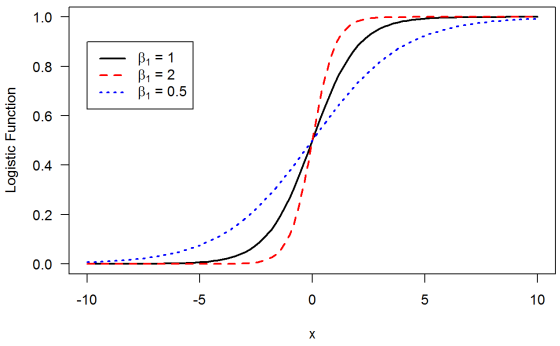


Figure 2.4.4: Different values of $beta_1$ in the logistic function

This two pictures show how the logistic regression fitting works, by varying the value of the parameters also varies the curve which results every time in different logistic functions. Once found the optimal parameters that maximize equation 2.30 then it has been found also the resulting logistic function in Figure 2.4.5 for the actual data.

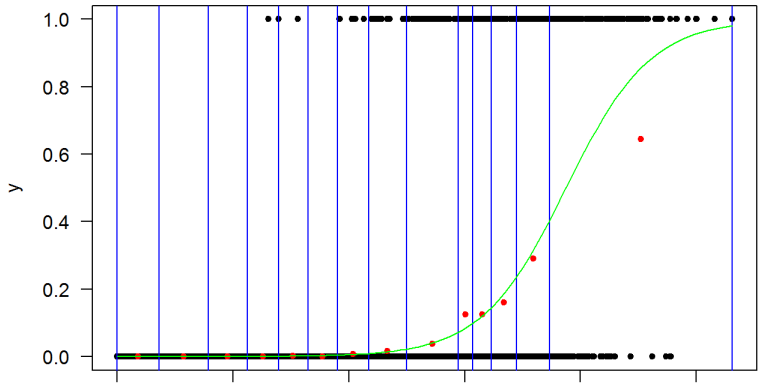


Figure 2.4.5: The green curve represents the optimal logistic function, where the red points are the new data to which the curve aims to classify with the resulting probability

According to [27], there are many approaches to know the goodness of the fitted model. McFadden (1973) suggested an alternative known as 'likelihood ratio index', which consists in comparing a model without any predictors to a model including all predictors. Because a binary response variable will not be normally distributed and because the form of the relationship to the binary variable will tend to be nonlinear it would be inconsistent to use a normal R^2 for evaluating the model. Instead, there is a pseudo R^2, proposed by Mc Fadden (1974), known also a Mc Fadden's R^2 and it is calculated as follows:

$$R^2_M = 1 - \frac{ln(L_M)}{ln(L_0)} \tag{2.36}$$

In this case L_0 is the likelihood function computed for a model with no predictors (i.e. considering the only paramter β_0) and L_M is the likelihood for the estimated model. L_0 plays an important role analogous to the residual sum of squares in linear regression, consequently, this formula corresponds to a proportional reduction in "error variance".
Another similar approach concerns on the Cox and Snell formulation, [28]:

$$R^2_{C,S} = 1 - (\frac{L_M}{L_0})^{2/n} \tag{2.37}$$

Since the equation 2.37 can be naturally extended to other kinds of regression estimated by maximum likelihood, then it's more appropriate to describe this as a "generalized" R^2 rather than a pseudo R^2. On the other hand, it has the problem with the resulting upper bound that is less than 1.0, more specifically is $1 - L_0^{2/n}$.

2.5 Methods based on Bayes techniques

Bayes' theorem is of fundamental importance for inferential statistics and many advanced machine learning models. Bayesian reasoning is a logical approach to updating the probability of hypotheses in the light of new evidence, and it therefore rightly plays a pivotal role in science, [29]. Bayesian analysis allows us to answer questions for which frequentist statistical approaches were not developed. In fact, the very idea of assigning a probability to a hypothesis is not part of the frequentist paradigm, [30]
According to [18], suppose that the goal is to classify an observation into one of K classes, where $K \geq 2$. Let:

- π_k representing the overall or prior probability that a randomly chosen observation comes from the prior kth class
- $f_k(X) = Pr(X|Y = k)$ denoting the density function of X (which is now a random variable) for an observation that comes from the kth class.

The Bayes' theorem states that:

$$Pr(Y = k|X = x) = \frac{\pi_k f_k(x)}{\sum_{l=1}^{K} \pi_l f_l(x)} \tag{2.38}$$

$Pr(Y = k|X = x)$ is called *posterior probability* that an observation X = x belongs to the k class.
Instead of directly computing the posterior probability $p_k(x)$ as in equation [citaeqn], we can simply plug in estimates of π_k and $f_k(x)$ into 2.38. Since it possible to compute the fraction of the training observations that belong to the kth class, then the value of π_k is straightforward. Since the density function $f_k(x)$ much more difficult to compute, it could be estimated, based on assumptions, in three different ways, [18].

1. Linear Discriminant Analysis (LDA)
2. Quadratic Discriminant Analysis (QDA)

3. Naive Bayes

Assuming that $f_k(x) \sim \mathcal{N}$ and also assuming that for simplicity the problem has only one predictor, the density function can be estimated using *LDA*:

$$f_k(x) = \frac{1}{\sqrt{2\pi\sigma_k^2}} \exp\left(-\frac{1}{2\sigma_k^2}(x-\mu_k)^2\right) \tag{2.39}$$

Where μ_k and σ_k^2 are the mean and variance parameters for the kth class. Plugging 2.39 into 2.38 it is possible to classify a single observation, according to the greatest $p_k(x)$.
With the same logic the equation 2.39 can be extended also to the case of multiple predictors, [18] where now $X = (X_1, ..., X_p)$ comes from a multivariate gaussian distribution with a mean vector $E(X) = \mu$ and a common covariance matrix $Cov(X) = \Sigma$, $(p \times p)$:

$$f(x) = \frac{1}{(2\pi)^{p/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right) \tag{2.40}$$

Once again, [18], after plugging 2.40 into 2.38 and doing a bit of manipulation, the Bayes classifier assigns an observation to the class k for which $\delta_k(x)$ is the greatest, where:

$$\delta_k(x) = x^T\Sigma^{-1}\mu_k - \frac{1}{2}\mu_k^T\Sigma^{-1}\mu_k + \log(\pi_k) \tag{2.41}$$

Regarding *QDA*, it assumes that each class has its own covariance matrix, making the final decision $\delta_k(X)$ different from the *LDA* generalized method, where:

$$\delta_k(x) = -\frac{1}{2}x^T\Sigma_k^{-1}x + x^T\Sigma_k^{-1}\mu_k - \frac{1}{2}\mu_k^T\Sigma_k^{-1}\mu_k - \frac{1}{2}\log|\Sigma_k| + \log(\pi_k) \tag{2.42}$$

Finally the last method for estimating the density function is called *Naive Bayes*.
Rather than assuming that these functions belong to a particular family of distributions *Naive Bayes* makes a single assumption, [18]:
"Within the kth class, the p predictors are independent", where the density function is given by:

$$f_k(x) = f_{k1}(x_1) \times f_{k2}(x_2) \times \ldots \times f_{kp}(x_p) \tag{2.43}$$

Since assuming that the p covariates are independent within each class, then there is no reason to worry about the association between the p predictors, because of the assumption that there is no association between the predictors.
One more time, plugging 2.43 into 2.38 it can be obtained an expression for the posterior probability, computed for each class k:

$$p_k(x) = \frac{\pi_k \cdot f_{k1}(x_1) \cdot f_{k2}(x_2) \cdot \ldots \cdot f_{kp}(x_p)}{\sum_{l=1}^{K} \pi_l \cdot f_{l1}(x_1) \cdot f_{l2}(x_2) \cdot \ldots \cdot f_{lp}(x_p)} \tag{2.44}$$

A characteristic problem with this method is that if a particular attribute value does not occur in the training set in conjunction with every class value, then things go wrong, [31].
A simple example could be the following: suppose that the model learn from the training

set of Table 2.3.1 that the value *play*: *rainy* is always associated with the outcome *No*. Then $Pr[Play = rainy|Yes] = 0$ because from the equation 2.43, the other probabilities are multiplied by the final probability.
As a consequence of that, if the the test set contains this observation, which is not present in the training set, the method would give a final probability of zero. This bug is easily fixed by minor adjustments to the method of calculating probabilities from frequencies with a method called *Laplacian estimator*.
Assuming in this scenario that the training set is so large that adding one to each count that would only make a negligible difference in the estimated probabilities,[32] one possible solution is to smooth all probabilities upwards by a count of 1.
This is now a fully Bayesian formulation, with prior probabilities assigned to everything in sight.
Eventually the pros and cons of adopting this method could be summarized:

- High accuracy and speed when applied to large databases.
- Simple and easy to understand algorithm, making it a good choice for rapid prototyping and baseline
- "Naive" assumption that features are conditionally independent. Actually, many dataset have correlated features, and this assumption can lead to suboptimal performance.
- Due to its simplicity, Naive Bayes may not capture complex relationships in the data as effectively as more advanced algorithms, such as decision trees or neural networks.

2.6 Support vector machine

Support Vector machines (SVM) can be defined as systems which use hypothesis space of a linear functions in a high dimensional feature space, [33]. SVM is actually a generalization of a simple and intuitive classifier called *maximal margin classifier*.

The underlying methodology is taken following [18].

2.6.1 Maximal margin hyperplane

First of all is important to define an hyperplane. In a p dimensional space, a hyperplane is subspace of dimension $p - 1$.
The mathematical definition of hyperplane to p dimension is given by:

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_p X_p = 0 \tag{2.45}$$

Now, if a point $X = (X_1, ..., X_p)$ do not satisfy the equation 2.45, (making it greater or less than zero) then X will not lie on the hyperplane, but on the other side of it.
A single point in the feature space belongs to a specific class, where here for simplicity the problem is constituted from 2 classes. Each point is classified into 2 classes according to $y_1, ..., y_n = -1, 1$.

Assuming that the hyperplane separates the n training observations perfectly according to their labels, the separating hyperplane for $i = 1, ..., n$ can be expressed by:

$$y_i(\beta_0 + \beta_1 X_{i1} + \beta_2 X_{i2} + \ldots + \beta_p X_{ip}) > 0 \tag{2.46}$$

Then, it is possible to compute the label of the point with sign of $f(x*) = \beta_0 + \beta_1 X_1 * + \beta_2 X_2 * + \ldots + \beta_p X_p *$, where:

- $f(x*) > 0$ the point i will belong to the class $y_i = 1$
- $f(x*) < 0$ the point i will belong to the class $y_i = -1$

Further is $f(x*)$ from 0, more the point x* lies far from the hyperplane and so more the point it is classified according to high level of confidence. Otherwise if $f(x*)$ is close to zero, the point would be classified always in the same class, (if the sign of 2.46 is the same for both), but with a low level of confidence, since it is closer to the separated hyperplane. As the Figure 2.6.1 shows, it is possible to have various hyperplanes that perfectly splitted the points in 2 classes, but with different values of $f(x*)$. As a consequence the best hyperplane that best divides the observations, is called *maximal margin hyperplane*, which is selected according to the one which is the furthest from the training observations, as shown in Figure 2.6.2

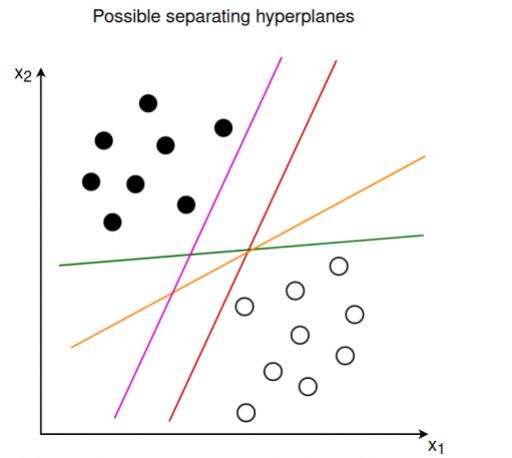


Figure 2.6.1: Different possible hyperplanes for n observations, [34]

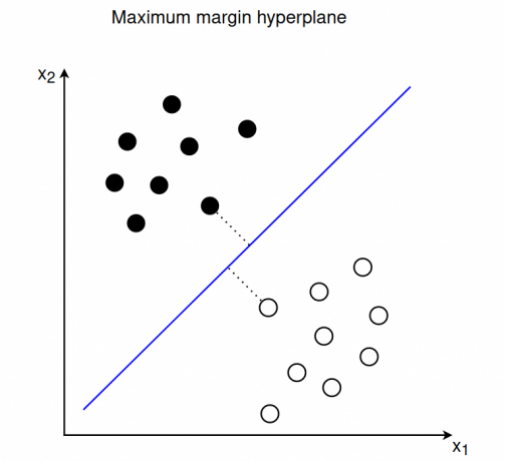


Figure 2.6.2: Selecting the best hyperplanes for the observations, [34]

The analytical solution about this problem is given from the following optimization problem, where the maximal marginal hyperplane (M) is the final solution.

$$\underset{\beta_0,\beta_1,\ldots,\beta_p,M}{\text{Maximize}} \quad M \tag{2.47}$$

$$\text{subject to} \quad \sum_{j=1}^{p} \beta_j^2 = 1, \tag{2.48}$$

$$y_i(\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \ldots + \beta_p x_{ip}) \geq M \quad \forall i = 1, \ldots, n. \tag{2.49}$$

2.6.2 Soft margin hyperplane

Actually, in many cases a perfect separating hyperplane does not exist, and so there is no M that maximize a perfect potential hyperplane among others. In fact there is no solution for this optimization problem with $M>0$. In addition there is also a risk that the training data could be overfit.
For this reasons, it is preferable to introduce a tuning parameter that allows an hyperplane that lead to a better classification in the test set. In this way it is tolerable that some observations will end up on the wrong side of the hyperplane even though this mechanism will result in a single classification error.
Eventually the previous optimization problem is integrated with the tolerance to accept some misclassification error:

$$\underset{\beta_0,\beta_1,\ldots,\beta_p,M,\epsilon_1,\ldots,\epsilon_n}{\text{Maximize}} \quad M \tag{2.50}$$

$$\text{subject to} \quad \sum_{j=1}^{p} \beta_j^2 = 1, \tag{2.51}$$

$$y_i(\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \ldots + \beta_p x_{ip}) \geq M(1 - \epsilon_i) \quad \forall i = 1, \ldots, n. \tag{2.52}$$

$$\epsilon_i \geq 0, \qquad \sum_{i=1}^{n} \epsilon_i \leq C \tag{2.53}$$

Where:

- M is the width of the margin to maximize s.t. the constrains

- C is non-negative tuning parameter that determine the severity of the violations to the margin M.
  Large values of C produce a smaller margin, trying to classify more observations correctly. On the other hand small value of C will produce a larger-margin, even if misclassifying some points.
  Then C is then chosen through a cross-validation in order to find a good balance between the bias-variance trade-off.

- $\epsilon_1, ..., \epsilon_n$ are called *slack variables* that allow the observations to be on the wrong side of the margin.

The solution for this optimization problem (support classifier problem) is given by only the inner products of observations, more specifically:

$$f(x) = \beta_0 + \sum_{i=1}^{n} \alpha_i \langle x, x_i \rangle \tag{2.54}$$

Because of their importance in changing the support vector classifier and so the margin M, observations that lie directly on the margin, or on the wrong side of the margin for their class, are known as *support vectors*.

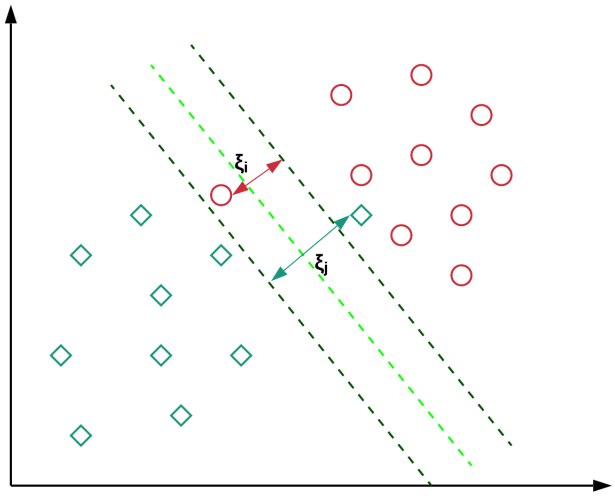
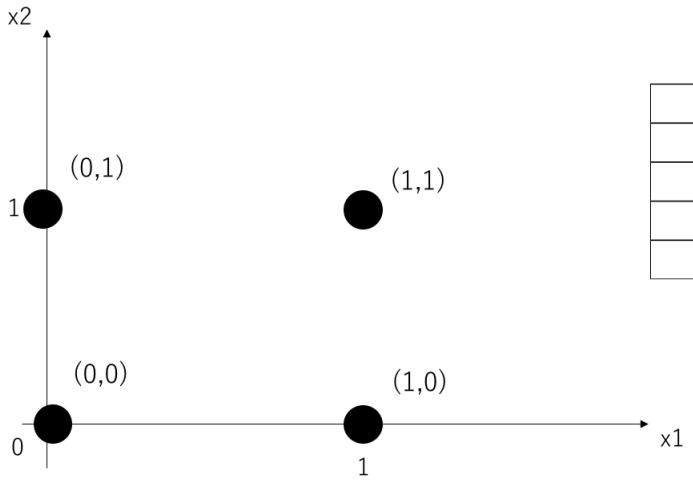


Figure 2.6.3: Soft margin hyperplan in the green

2.6.3 Kernel machines

Since the last model presented is a **linear** model, meaning that aims to partitions the feature space through straight lines, it so not sufficient since in many cases the relationship between features and the response variable is **non linear**, such as many real-world problems, as shown in Figure 2.6.4.



Coordinate	Class
(0,0)	1
(0,1)	0
(1,0)	0
(1,1)	1

Figure 2.6.4: XOR problem, [35]

In order to overcome this problem, it has been introduced the **Kernel machines** that extend the SVMs to the non linear case, [36].
Since the equation 2.54 has n parameters α that are different from zero only for the points in the training set known as support vector, this equation could be written also as follows:

$$f(x) = \beta_0 + \sum_{i \in S} \alpha_i K(x, x_i) \tag{2.55}$$

Where:

- S is the collection of indices of the support vector points
- K is some function that quantifies the similarity of two observations, known as *Kernel*, such as:
  Polynomial kernel: $K(x_i, x_{i'}) = \left(1 + \sum_{j=1}^{p} x_{ij} x_{i'j}\right)^d$, where p is the number of features in the dataset and d is the degree of the polynomial
  Radial kernel: $K(x_i, x_{i'}) = \exp\left(-\gamma \sum_{j=1}^{p} (x_{ij} - x_{i'j})^2\right)$, where γ is a positive constant.

In order to a better understanding of the importance of kernel machines, the Figure 2.6.5 shows a classification .

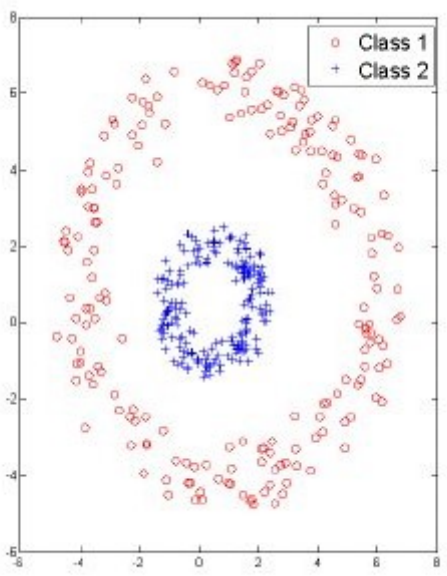


Figure 2.6.5: Classification of Support vector machines based on a radial kernel, [37]

2.7 KNN

The k-nearest neighbors (KNN) algorithm is supervised ML algorithm, which is simple, easy-to-implement and can be used to solve both classification and regression problems. Then the basic idea is to assign an individual to the the population whose sample contains the majority of classes from the k 'nearest neighbours', [38].
In a few words, KNN algorithm classify a point based on the *similarity* which this point has with the rest of population.
This concept of similarity is expressed by the distance metric, which is euclidean by default.
The distance can be defined also in different ways. Here for simplicity these metrics refer to a dataset with 2 features, x_1, x_2:

- Manhattan distance: $\sum_{i=1}^{N} |x_{1i} - x_{2i}|$
- Minkowsky distance: $(\sum_{i=1}^{n} |X_1 i - x_2 i|^p)^{\frac{1}{p}}$, where is a parameter.

How KNN works can be better understood from a pseudo-algorithm:

1. Choose an hyper-parameter k
2. Compute the euclidean distance d from the observation point to the other N points: $d = \sqrt{(x_1 obs - x_1^i)^2 + ... + (x_p obs - x_p^i)^2} \qquad \forall i = 1, ..., N$
3. Sort in a list in ascending order the first k points according to the distances computed
4. Count the votes for each class (label) among the neighbors

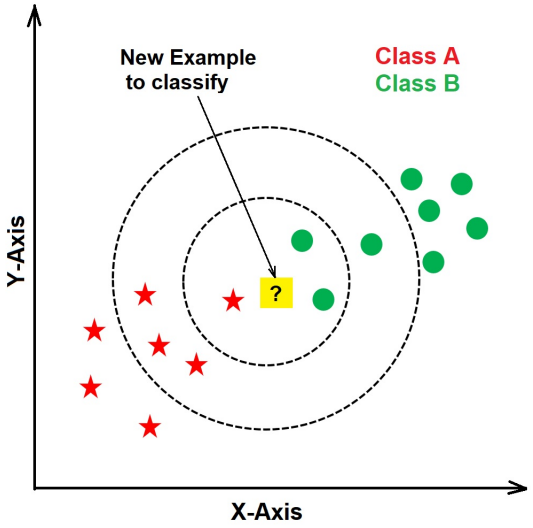


Figure 2.7.1: KNN Classification

5. Classify the observation point $x_{obs} = (x_1 obs, ..., x_p obs)^T$ according to the class that "gets the most votes" within the first k in the list, (if it is a regression problem, it returns the average of the k points considered).

Since the KNN algorithm does not learn anything because it just classify a point based on the distance, it is not defined properly as a learning algorithm. In the dataset related to Figure 2.7.1, there are only 2 features and a target variable that is responsible to assign a label to the point (red stars or green point here).
Always referring to this figure, it is possible to see how important the choice of K is: if $K = 3$ is chosen, then the point is classified as a green point (because of the majority of "votes" among the three points). Otherwise, if $K = 5$, the point is marked as a red star for the same reason.
Since KNN method is biased by k, as it suggest [39], there are many ways to choose the value of k, but a simple one is to use the cross-validation method, where the accuracy is measured with a different value of k each time. In this way, the highest value of k in the cross-validation will be the correct value to use in the test set.
Choosing the right value of k is made between these two considerations:

- small value of k means that noise and outliers will have a higher influence on the classification, resulting in a less stable prediction
- high value of k make computationally expensive as much as the number of features increase , being also against the basic philosophy behind the model, which aims that the observations that are closer to the point are more likely to have similar classes, (according to the concept of similarity based on the distance)

It is accepted that a good and simple approach, coming from a simple rule of thumb is to select $k = \sqrt{N}$, where N is the number of observations in the training data.
The KNN algorithm can struggle with imbalanced data, where one class has significantly more data points than the other, resulting in a biased classification results. From this problem it is possible to fix this issue assigning a weight to the neighbors, based on their distance.

In this case the data that belong to that particular class will be weighted and then "brought closer" to the neighbors, without being affected from this unbalancing.
Choosing an even value of k could end in a tie from the labels that comes from the most votes, in this case the best thing is to weight more the data closer the point to classify, as mentioned above.
Eventually KNN presents the main benefits and drawbacks:

- "curse of dimensionality" in high-dimensional space, meaning that the algorithm does not work well with an high number of features (the distance between observation will go to "collapse", making this metric not meaningful anymore)
- easy to implement
- it is no-parametric, which means it makes no assumptions about the underlying distribution of the data. This makes it a flexible algorithm that can be used in a wide range of applications, (has only the value k to fix from the user in advance, called hyper-parameter).

2.8 Artificial neural network

Since the 1940s, artificial neural networks (ANN) have been the subject of active research. As a component of connectionism (neural computations), ANN has progressed significantly from the period of unrealistic expectations, through the period of disappointment in the 1970s, to the current period of widespread application of the technology, [40].
The key behind this method is the connection between biological neurons and the possibilities of modelling them using logistic computations.
Many computationally challenging problems such as optimization, signal processing, image recognition, prediction and classification can be solved using a neural network model, [41].
A neural network takes an input vector of p variables $X = (X_1, X_2, ..., X_p)$ and builds a nonlinear function $f(X)$ to predict the response variable Y, [18].

2.8.1 Structure

Neural network are composed by nodes or units, arranged in 1 or more layers, which are connected by directed links. The layers involves the input features X, the target y and one or more hidden layer between these two.
The activation a_i is propagated from unit to unit via a link. In addition, each link has a numerical weight ω that indicates the direction and strength of the connection, [41].
The single-layer neural network, shown in Figure 2.8.1, is a neural network composed by three layers: one input layer (features), one output layer (target) and one *hidden layer* which is between them. This architecture of neural network is also called *perceptron* and can be described in two steps, [18]:

- For each unit $k = 1, ..., K$ in the hidden layers (only one layer here), compute the activation function A_k as a function of the input features $X_1, ..., X_p$.

$$A_k = h_k(X) = g(\omega_{k0} + \sum_{j=1}^{p} \omega_{kj} X_j) \tag{2.56}$$

- These K activations from the unique hidden layer are then fed into the output layer, resulting in:

$$f(X) = \beta_0 + \sum_{k=1} K\beta_k A_k \tag{2.57}$$

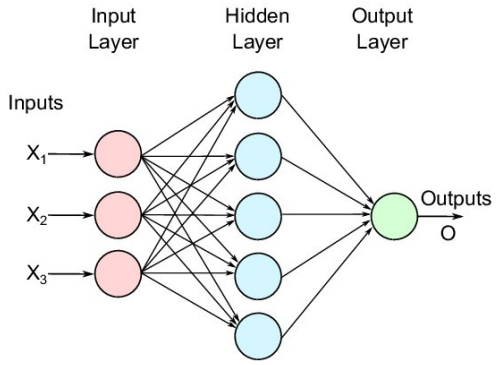


Figure 2.8.1: Structure of a neural network composed by one hidden layer, [42]

The activation functions are not computed in advance, but are fixed later during the training of the model, 2.62. Instead, the output layer uses these activations functions linearly as inputs, resulting in a function $f(X)$. Then, the final form of the neural network is:

$$f(X) = \beta_0 + \sum_{k=1}^{K} \beta_k g(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j) \tag{2.58}$$

Where estimation of the parameters $\beta_0, ..., \beta_k$ and $\omega_{10}, ..., \omega_{kp}$ is required. Instead, g is a non linear function which is described typically by a *logistic function* or a *ReLu* function (rectified linear unit), 2.59 and 2.60 respectively.

$$g(z) = \frac{e^z}{1 + e^z} = \frac{1}{1 + e^{-z}} \tag{2.59}$$

$$g(z) = (z)_+ = \begin{cases} K & \text{if } z < 0 \\ z & \text{otherwise} \end{cases} \tag{2.60}$$

As a consequence of these non-linearity functions, neural network can capture non-linearity relationship between features and target variable.
Once the mathematical model for individual "neurons" has been decided, the next task is to is to connect them together to form a network. There are two fundamentally different ways to do this, a **recurrent network**, which feeds its output back back into its own inputs and a **feed-forward network**. A feed-forward network forms a directed acyclic graph because it has connections that only point in one direction. Here there are no loops, each node passes information from "upstream" nodes to "downstream" nodes, [18] . Feed-forward networks are typically constructed in layers, with each unit only receives input from units in the layer immediately before it and returns an output to all the units in the next layer.

2.8.2 Model training and backpropagation

Eventually, given an observation (x_i, y_i) where x_i represents an entire set of features data $x_1, ..., x_p$ and y_i the response, it is possible to develop a fitting model in the following way, [18]:

$$\underset{\omega_k,\beta}{\text{minimize}} \frac{1}{2}\sum_{i=1}^{n}(y_i - f(x_i))^2 \tag{2.61}$$

$$\text{where} \quad f(x_i) = \beta_0 + \sum_{k=1}^{K}\beta_k g(w_{k0}) + \sum_{j=1}^{p} w_{kj}x_{ij}^2 \tag{2.62}$$

This optimization problem is non-convex, meaning that has multiple feasible regions and multiple locally optimal points within each region. Depending on the number of variables and constraints, it can take an exponential amount of time to determine an optimal solution. For this reason 2 strategies are applied to overcome this issue:

- Slow learning: Gradient descent is used to iteratively fit the model. The fitting model is terminated when overfitting is detected.
- Regularization: in this fitting process there penalties imposed on the parameters ω and β.

Involving all the parameters in one unique vector θ, the previous objective functions could be rewritten as:

$$R(\theta) = \frac{1}{2}\sum_{i=1}^{n}(y_i - f_\theta(x_i))^2 \tag{2.63}$$

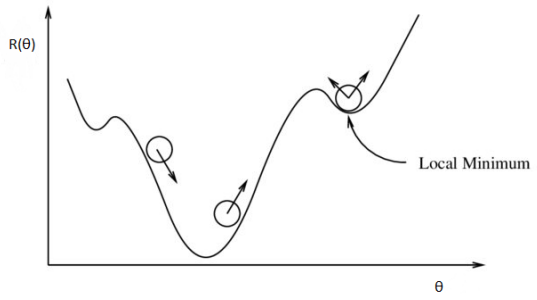


Figure 2.8.2: Gradient descent problem

The Figure 2.8.2 refers to the non-convex optimization problem of the gradient descent, which aims to find the global minimum taking as few steps as possible, resulting in an acceptable computation time.
At this time it essential to introduce the concept of **Backpropagation**, it is a process involved in training or fitting a neural network. It aims to take the value of the objective function (i.e. error of prediction) trying to minimize it for each iteration, called also *epoch*.
During the forward propagation the algorithm feed this *loss* backward through the neural network layers to fine-tune the weights, making the model more reliable increasing its generalization.
The idea behind this method is to find a direction of θ which brings the objective function

to its *global minimum*, even tough is not guaranteed.
The local minimum point is that particular point where the all parameters needed for building the objective function, and so the neural network, assume their best value in order to make the final prediction.
This is made computing the direction where $R(\theta)$ increases most rapidly and then to move θ in the opposite direction, where m here represents his value after a single iteration, [18]:

$$\nabla R(\theta_m) = \left.\frac{\partial R(\theta)}{\partial \theta}\right|_{\theta=\theta_m} \tag{2.64}$$

Since $R(\theta) = \sum_{i=1}^{n} R_i(\theta)$ then:

$$R_i(\theta) = \frac{1}{2}\sum_{i=1}^{n}\left(y_i - \beta_0 - \sum_{k=1}^{K}\beta_k g_k(w_{k0} - \sum_{j=1}^{p} w_{kj}x_{ij})\right)^2 \tag{2.65}$$

Where it could be simplified using $z_i k = w_{k0} - \sum_{j=1}^{p} w_{kj}x_{ij}$:

$$R_i(\theta) = \frac{1}{2}\sum_{i=1}^{n}\left(y_i - \beta_0 - \sum_{k=1}^{K}\beta_k g_k(z_i k\right))^2 \tag{2.66}$$

Eventually, applying the differentiation respect to both parameters β, $\omega_k j$ will result in the best direction of θ to reach the local minimum point:

$$\frac{\partial R_i(\theta)}{\partial w_{kj}} = -(y_i - f_\theta(x_i)) \cdot \beta_k \cdot g(z_{ik}) \cdot x_{ij} \tag{2.67}$$

$$\frac{\partial R_i(\theta)}{\partial \beta_k} = -(y_i - f_\theta(x_i)) \cdot g(z_{ik}) \tag{2.68}$$

2.8.3 Model boosting

As already mentioned, the purpose gradient descent algorithm is to move θ from the previous point θ^m towards the peak of the objective function, but oriented in the opposite direction, $-\nabla R(\theta_m)$. Then the this iterative method to find global minimum point can be expressed as:

$$\theta^{m+1} = \theta^m - \rho \nabla R(\theta_m) \tag{2.69}$$

Where ρ is an hyper parameter called *learning rate* which acts as a "boost" to help the function to find its global minimum, but tuning ρ is not so straightforward since:

- low value of ρ lead to going deeper into one of the local minimum points, but with the risk that it may be difficult to "get out". In this way there more possibilities to find best suitable points nearby, but the method could be trapped in one of these local minimum points with the risk of overfitting the training data.

- high value of ρ on the other hand aims to find the global minimum point by trying to search in a larger feasible region, thus having more possibilities to explore more points of the solution space, but with the risk of "skipping" some local minimum points.

Another hyper parameter regarding the implementation, is called *batch size*. Batch size refers to the number of examples from the training dataset used in the estimate of the error gradient and so the number of samples that will be propagated through the network. Since it captures the accuracy about the descending gradient method, it is possible to select among three different configurations about it, [43]:

- *Batch gradient descent*, where the batch size is the total number of observations in the training set.
- *Minibatch gradient descent*, where the batch size is set between one and the total number of examples in the training dataset. In this way the sample of a small fraction is representative to compute the gradient step. Its efficiency is better than the previous option since it does not store all training data in memory.
- *Stochastic gradient descent*, where the batch size considers only one random observation which is going to fit the *loss* function, (objective function in the optimization problem). The weight updates process is fast, but since it varies from one observation to another, it presents huge oscillations and then it is hard to retrieve the global minimum point, [44].

CHAPTER
THREE

METHODS

3.1 Drone experiment

During the work in the lab we have been provided, always from "Quanser" by a drone called 'Qdrone 2', showing in Figure 3.1.1.

Figure 3.1.1: QDrone 2

The experiments with it were forced to conclude since we realised that the drone presented some failures at one oh his propeller engine, making the device unstable at every run of the experiment making us lose his control, as shown in Figure 3.1.2 and 3.1.3.
In Figures 3.1.2, 3.1.3 we can see the *Roll, pitch and yaw* angles, that are terms commonly used in aviation, aerospace and robotics to describe the rotational movements of an object. Unfortunately in this very short simulation with, lasting just a few seconds, the drone started to lose his controls after its take off. Eventually, as soon as we realised the risk of the simulation, the experiment was immediately stopped.

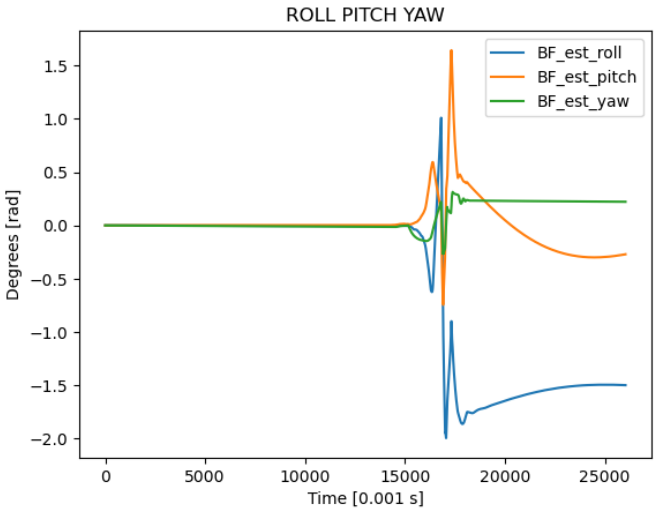


Figure 3.1.2: Roll pitch and yaw degrees during the simulation

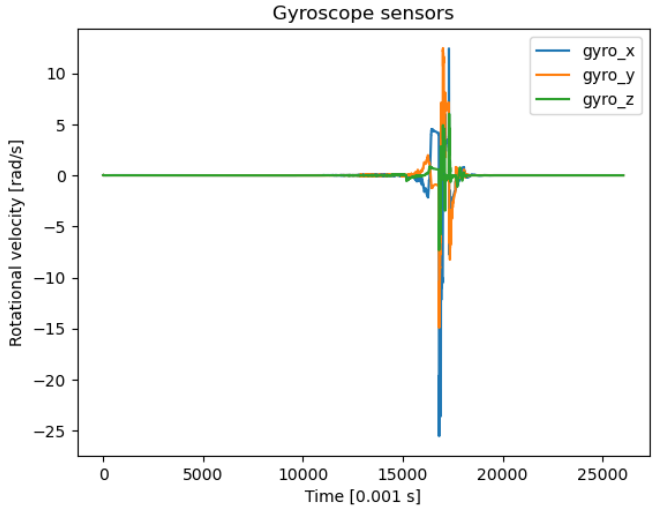


Figure 3.1.3: Gyroscope sensors during the simulation

3.2 Experiment description

Firs of all, the software that has been used from the company to develop and integrate the robot is called "QUARC". QUARC generates real-time code directly from Simulink-designed controllers and runs it in real-time on the Windows target, [3]. This makes it possible to develop, deploy and validate real-time applications on hardware using Simulink.
First, the Qbot Simulink model, which is a Digital twin of the device, was modelled to obtain meaningful signals that could be functional in identifying collisions. These signals were then transformed into MATLAB variables for the next steps.
Since the robot collision was detected by all of three bumper sensors (left, centre and right), our purpose was to a ML model that aimed to predict the right a real-time collisions in order to completely replace or substitute the need of these sensors[1].

[1] *Qbot, robot, device* were used with the same meaning

The first step was to collect a meaningful dataset in order to simulate the different uses that the robot might be subjected in future scenarios.
The simulation has been set up to collect data from the robot every 0.01 seconds, meaning that every 0.01 seconds the Digital twin of the Qbot Simulink model collects one signal from different sensors, (i.e. one row of the dataset) .
Then, We obtained a dataset that was representative of a 20 minute simulation in order to recreate every possible behaviour of the robot [2]. Python was then used to process and analyze this dataset.
The dataset was then processed through various ML algorithms to select the best candidate model with the highest score that best replicated the actual output of the bumper sensor.
Once we have computed all the score for every model, the best model selected turned out to be an artificial neural network (ANN) built using 'Keras' library.
In the table 3.2.1 we can see the signals used as features to predict the binary target variable called 'BUMP', where:

$$y(t) = BUMP_t = \begin{cases} 1 & \text{if the Qbot hit something at the istant } t \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

ang_acc	gyro	lv_actual	lv_comm	lwheel	lwheel_acc	rv_actual	rv_comm	rwheel	rwheel_acc	BUMP
0.000	2.597	0.157	0.185	0.212	0.206	-0.473	-0.625	0.101	0.014	0
-0.960	2.587	0.156	0.185	0.212	-0.023	-0.476	-0.617	0.100	-0.080	0
0.000	2.587	0.157	0.185	0.214	0.158	-0.481	-0.617	0.101	0.028	0
0.000	2.587	0.157	0.185	0.213	-0.063	-0.481	-0.617	0.100	-0.068	0
-1.379	2.574	0.155	0.185	0.211	-0.244	-0.477	-0.617	0.099	-0.146	0
-0.873	2.565	0.154	0.185	0.211	-0.014	-0.478	-0.617	0.098	-0.040	1
0.000	2.565	0.155	0.185	0.212	0.159	-0.484	-0.617	0.099	0.032	1

Table 3.2.1: Dataset containing some values of all the features used to build the ML models

We can summarize the meaning of the features we used:

Table 3.2.2: QBot 2e specifications

Name used	**Meaning**
ang_acc	angular acceleration
gyro	gyroscope signal which is used to estimate the robot's orientation and the yaw angle
lv_actual, lv_comm	linear velocity of the commanded (input) data from the user and the actual (output) data from the robot
lwheel, rwheel	linear velocity of the left and the right wheel
lwheel_acc, rwheel_acc	left wheel and right wheel acceleration
rv_actual, rv_comm	rotational velocity of the commanded (input) data from the user and the actual (output) data from the robot
BUMP	bumper sensors

[2]By the term *behaviours*, we refer to all the possible movements and speeds it can perform

First of all it is interesting to focus on 2 different signals and their interpretation: Regarding the signals characterised by *commanded* and *actual*, they are related respectively to the input signal given by the user (from the joystick) and the output signal coming from the Qbot.
The first one comes from the Kinematic model and describes how the robot's position and orientation change over time in response to control inputs, such as wheel velocity, without considering the underlying physics of the robot. Instead, the *actual* signal comes from the *dead reckoning* model and so is based on odometric data. The difference refers to the dynamic circumstances that happen while the robot is interacting with the environment, making the commanded signals slightly different from the actual one.

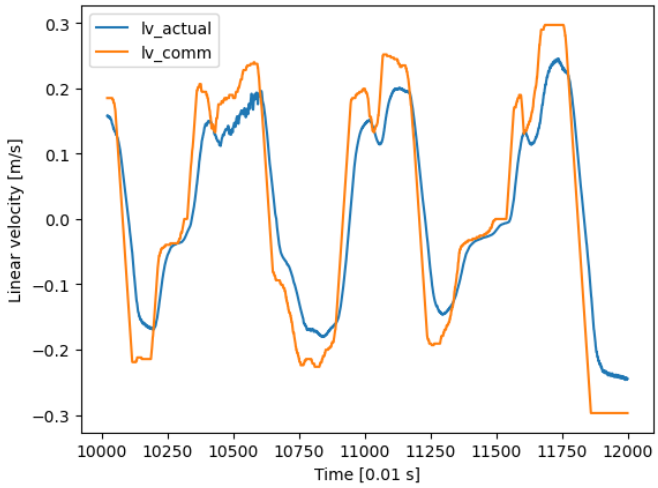


Figure 3.2.1: Commanded and actual linear velocity

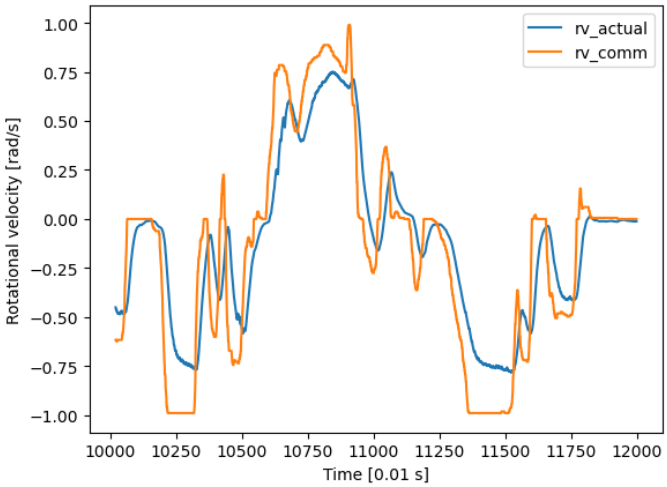


Figure 3.2.2: Commanded and actual rotational velocity

As we can see in Figure 3.2.1 and 3.2.2 the goal of controlling system is to make the difference between commanded and actual speed close to zero.
On the other hand, the gyroscope aims to estimate the robot's orientation or yaw angle, which is the angle XY, as shown in Figure 3.2.3. It can detect any sudden changes in the robot's angular velocity, allowing it to make quick adjustments to avoid tipping over

or losing its balance. Gyroscopes can also compensate for external forces acting on the robot, such as vibrations or sudden impacts. This helps to maintain an accurate orientation and minimizes deviations from the intended path, corresponding to a "pulse" acceleration that is detectable by it.

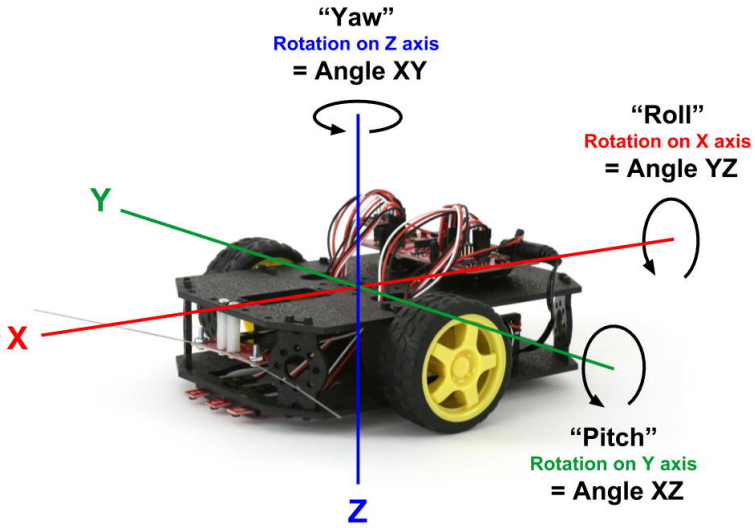


Figure 3.2.3: Visualization of roll, pitch and yaw angles on a device, [45]

Taking the derivative $\frac{d(\text{gyro})}{d(time)}$ we obtain an accelerometer that could be useful to detect when the robot is physically bumped by changing in motion.
Since this is a binary classification problem, we moved the data to Python and we started to run different appropriate ML models [3]:

- Decision Tree
- Logistic Regression
- Naive Bayes
- KNN
- Artificial neural network

All the models have been run using 'scikit-learn' library, except for the ANN model, which we used 'Keras'. These models have been run several times, each time with different parameters related to the model itself. Concerning the Decision tree we tested it with different values of *max_ depth number of leaves*. Afterwards in order to prune the tree, we also selected the best α to evaluate the cost complexity pruning.
On the other hand for running KNN we tried different *number of neighbors*.
Regarding the ANN we selected each time different random values of *hidden layers* and *number of neurons*.
After fitting each model to the training set and calculating a 'score' for each based on the test set, we selected an ANN as the model that best predicted the variable $BUMP$.

[3] It was decided to standardize the data before running the KNN and SVM models, as they are more sensitive to outliers

In order to pass the ANN model to Simulink, we had to find a suitable way of doing this so that it could actually be read and then managed to work during the simulation.
For this purpose, we decided to save and store the final ANN model in the ".hdf5" format. It is possible to briefly summarise the uses and advantages of this format:

- It is designed to organize and manage data efficiently, making it suitable for storing both the architecture (model structure) and weights (parameters) of deep learning models.
- Increased interoperability with applications, providing a versatile choice for storing and sharing ML models in different environments, as it is compatible with different programming languages such as Python and MATLAB, [46].

Furthermore, various experimental data-driven ML approaches have been successfully applied to solve different problems, such as data-driven physical models, [47].
Since in our work there was a need for greater **interoperability** in the AI tools community, ONNX defines a common set of operators which involves the development of ML and deep learning models. ONXX is common file format to enable AI developers to use models with different frameworks, tools, runtimes, and compilers; exploiting also libraries designed to maximize performance across hardware. [48].
As interoperability was a key requirement in our work, it was important how ONXX integrates different software, such as Keras and MATLAB, and allows them to communicate with each other. For this reason we took advantage of using *'Deep Learning Toolbox Converter for ONNX Model Format'*, which was able to exploit the trained model file (ANN.h5).
Then we also installed the *'Deep Learning Toolbox Converter for TensorFlow Models'*, which allows us to import a previously trained TensorFlow model with its weights from Python to Simulink.
The next step was to modify the Digital twin Simulink model environment so that it could host the ANN that we had created.
Since the Qbot model in Simulink had exactly the hardware device as its "target", it was necessary to act on QUARC blocks to find the right path for our purposes.
To pursue this, it was first created a Simulink file, acting as a *server*, where through specific blocks capable of hosting the ANN model it was possible to elaborate signals in real-time (i.e. every 0.01 s) during the simulation; while the default one acted as *client*.
The "Stream Client" QUARC block acts as the client in the communication process, connecting to a local or remote host and sending and/or receiving data from that host.
On the other hand, the "Stream Server" QUARC block acts as the server in the communication process, it listens for and accepts a connection from a local or remote host and sends and/or receives data from that host, [49].

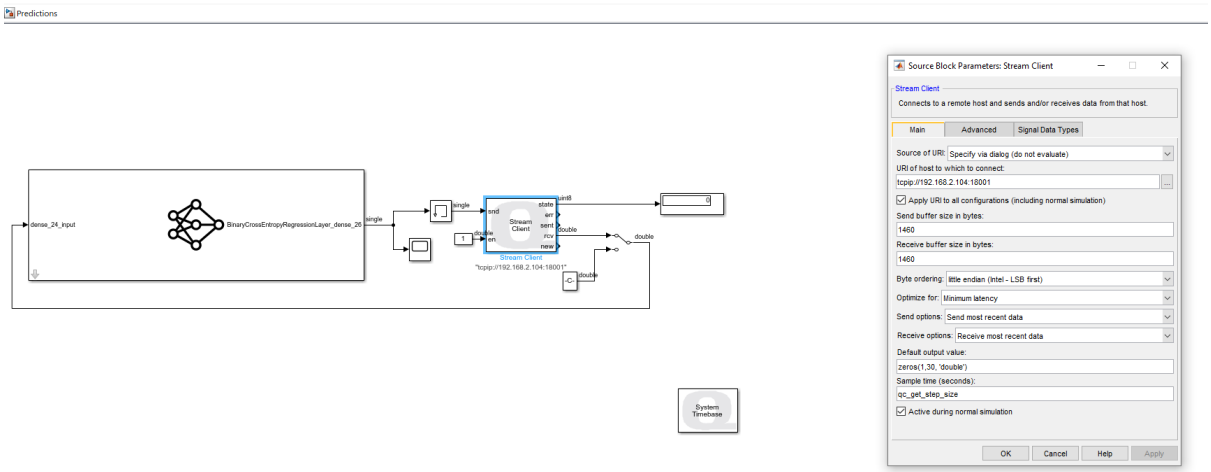


Figure 3.2.4: Stream Client instance which involves the output of the ML model

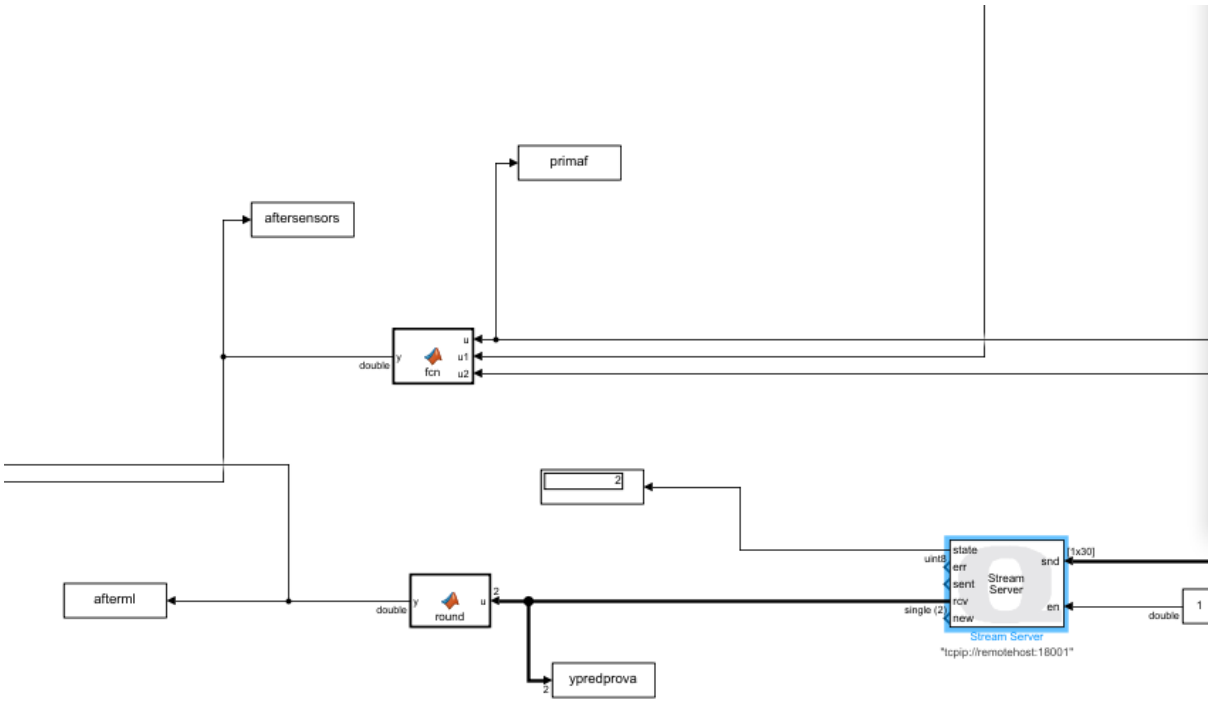


Figure 3.2.5: Stream Server instance on the default model

The Figure 3.2.5 shows the **Stream Server** block that receives as input the signals (features in our ML model) that were occurring during the simulation and then sends them to the **Stream Client** block.
The purpose of the client block is to take them as input to the ANN model, send them to the model which is present to the same file and eventually pass the final output back to the server block (actually the roles of client and server are reversed in this case).
The signals taken as input of the ML model are represented by the first 10 columns of the table 3.2.1.
This allowed the output from the ML model to be compared in real time with the reference output from the sensor, thus obtaining:

1. Bumper sensors output, coming from the default Simulink model
2. ML model output

As a consequence of acting on the last point, the goal was to make the difference between them as close to zero as possible.
Finally, it was possible to create and modify an empirical threshold placed after the ML model to adjust the probability according to which the current value is classified, as better explained in the next section. This final threshold is created with a MATLAB function and it is basically an if-else condition. It refers to the output of the ML model according to which the values will be classified with a probability.
The aim of it was the following: if during or after the simulation we realised that the ML model is too sensible, ending with high percentage of false alarm (detecting too many collisions) we could change the final threshold by increasing or decreasing it in order to allow a better collision detection during the simulation.
Finally, the Digital virtual twin model of the QBot was ready to detect the collision and also to estimate its duration, taking advantage of the sensors on the one hand and the ML model on the other.
Here, in Figure 3.2.6 it is possible to see the corresponding experimental framework according to what we have done:

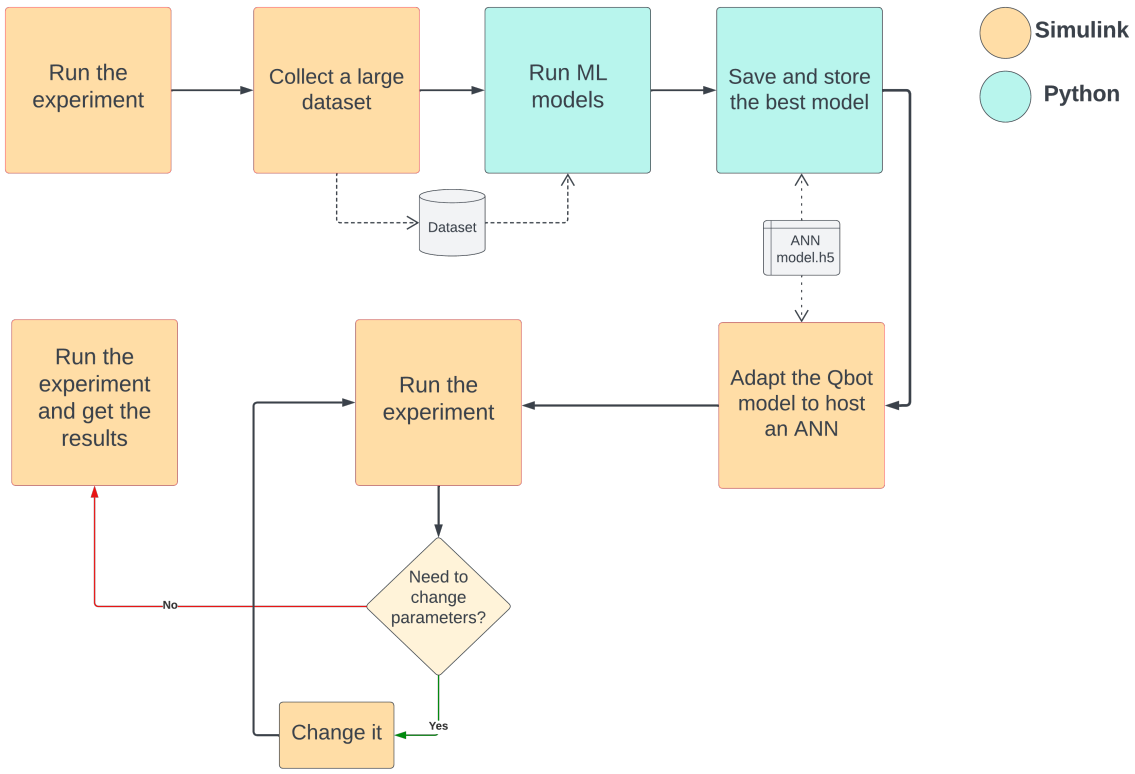


Figure 3.2.6: Flowchart representing a conceptual framework of the work that has been carried out

As introduced in the previous chapter, during the neural network fitting we want to minimize the objective function of the actual problem.
Depending on the actual problem, some objective function, called also *Loss function*, are

more appropriate than others. In this case, in order to monitor the performance during the model fitting, we used a *binary cross-entropy* loss function, which is described as follows, [50]:

$$-\frac{1}{N}\sum_{i=1}^{N}[y_i \log(p(y_i)) + (1-y_i)\log(1-p(y_i)] \tag{3.2}$$

Where:

- N is the total number of observations in the training set
- y_i is actual target of the ith observation
- $p(y_i)$ is the probability that the related class is classified as "1"

3.3 ANN Architecture

In order to build the best ANN possible, we tried to empirically change different parameters before fitting the model, such as:

- number of hidden layers
- number of units for each hidden layers
- learning rate
- batch size

At the end, after many attempts, the neural network model which gives the best performance is the one shown in Figure 3.3.1.

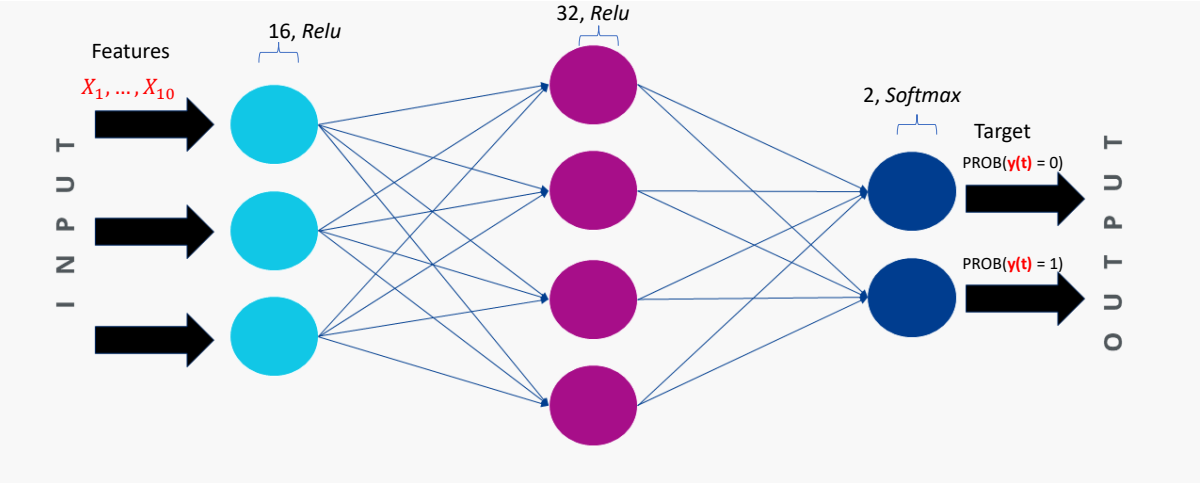


Figure 3.3.1: Architecture of the best neural network found

As we can see in Figure 3.3.1, the network is composed by 2 *ReLu* activation functions, composed by 16 and 32 units, respectively and 1 *Softmax* activation function which involves 2 units. For our scenario , softmax function has been the most suitable activation function before the output layer, as we shall see.

Since the purpose of this problem was fitting a neural network which had the goal to predict a binary classification response, the last activation function the was best suitable for this problem is called *Softmax* activation function. The softmax function is often used as the last activation function of a neural network to normalize the output of a network to a probability distribution over predicted output classes, [51]. It converts the previous weighted sum values to probabilities that sum to one, where each value in the output of the softmax function is interpreted as a probability of belonging to each class, [52].
Finally The output is a vector with probabilities of each possible outcome. The output probabilities sums to one for all possible outcomes or classes.
Softmax is defined as follows, [51]:

$$S(y)_i = \frac{e^{y_i}}{\sum_{j=1}^{n} e^{y_j}} \tag{3.3}$$

Where y_i comes as an input from the previous hidden layer and represent the likelihood to the ith class, while n is the number of classes. This means that for each observation we have as an output a vector consisting of 2 elements, each indicating the probability of belonging to class '0' and class '1' respectively.
In this scenario, as already mentioned, these output will represent the probability of each observation being "0" (no collision detected) and "1" (collision detected) respectively, according to the simulation time fixed at 0.01 [s]. Since we wanted to know only a pure classification for each observations, it was necessary to create an "artificial" layer before the output one which aims to create a simple threshold function.
Eventually, this threshold function aims to assign the final classification value $y(t)$ as follows [4] [5]:

$$\mathbf{y(t)} = \begin{cases} 1 & \text{if } S(y)_{i=1} >= 0.5 \\ 0 & \text{otherwise} \end{cases} \tag{3.4}$$

3.4 ANN Enhancement

Once we had reached this point, we asked ourselves whether it would be possible to make changes to the existing model in order to achieve a better final performance. Since it is reasonable that a potential collision could be affected not only from the actual signals, but also from the previous ones we came up with the following idea. Consequently, this binary classification problem could also be modelled with the features from the previous time step. As they could have an effect on the actual response variable, we focus on integrating the study analysis towards a time series problem.[6]

[4] $y(t)$ present in the equation 3.4 is not to be confused with y in the previous equation 3.3, where the first refers to the final classification outcome of the model

[5] Since we get new observations "every time" during the simulation, we will consider $y(t)$ as a function along time for simplicity

[6] Here, *target*, *response variable*, $y(t)$ are all synonyms used for the same thing

Later on, it was necessary to decide how many previous time steps we should take into account to improve the model. This implies that at each time t, the response variable $y(t)$ is computed not only by the features $X(t)$, but also by the features considered up to K time step earlier. We can visualize this potential improvement by comparing the differences between the existing ANN 3.5 to the new one, 3.6:

⋆

$$y(t) \xleftarrow{fitting} X(t) \tag{3.5}$$

⋆

$$y(t) \xleftarrow{fitting} X(t) + \sum_{k=1}^{K} X(t-k) \tag{3.6}$$

Where:

- $y(t)$ is the final outcome at the instant t
- $X(t) = X_1(t), ..., X_{10}(t)$ are the features at the instant t
- K refers to the maximum number of previous time steps considered, keeping in mind that each time step means 0.01 seconds

Then the second model written in 3.6 is a generalization of the first one; both models coincide when $K = 0$.
In this way, the new ANN score increased significantly, proving that the previously considered features did indeed have a strong influence on the actual target. The actual detection of a collision is therefore best predicted if we also take into account the previous signals.

Since we originally started with the first 10 features, each time an additional k is considered, the model is trained with 10*K features. So we always have the same type of signal, but for each k considered, the 10 features from the previous time step are added to the ML model as an additional signal for each observation during the simulation.
As a consequence for the model fitting, the first k rows of the dataset will consequently not be taken into account. However, this consideration is entirely negligible since the original length of the dataset was composed by 116.620 rows.
The following question that stands out is then what is the best value of K to take.
To answer at this question we then trained the neural network by considering an additional k at each iteration. The estimation of the best value was carried out mainly by considering the computational time taken to train the model and the actual improvement of the model from the previous one.

Having reached this point, however, it was necessary to make the Simulink model of the robot suitable for the neural network model in question. As a consequence, we used the 'Delay Time' block to simulate a delay of the current signals to be added as input to the neural network model in real time during the simulation.
Without taking into account the separation between the two actual Simulink models, composed by Client and Server, which is essential for the correct real-time execution of the neural network, the summary conceptual model (concerning only a part of the robot's digital twin) is shown in Figure 3.4.1.

This figure helps to give an idea of how the entire Digital Twin model works for $K = 1$ during the simulation:

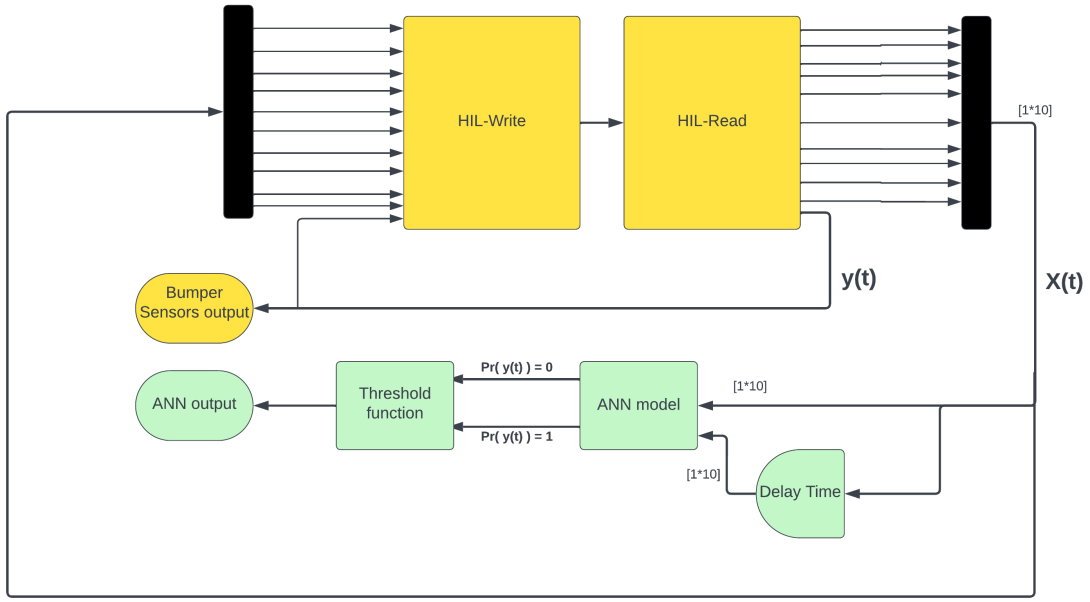


Figure 3.4.1: Conceptual Simulink flowchart

Where:

- The HIL-Write block represents the interaction and control with the robot itself from input signals.
- The HIL-Read block represents the possibility of storing output signals and then possibly modifying them

In this way it is possible to also artificially modify the behaviour of the robot, for example to make it turn back when it hits an obstacle, it would be enough to create intermediate functions and link these two blocks together. In this scenario, the yellow blocks represent the standard blocks that already exist, while the green blocks represent the blocks that have been added to properly accommodate and validate our neural network model.
Finally, given the strong imbalance of the dataset between the two classes the undersampling technique was implemented, which, however, did not prove effective in predicting collisions in real time. It was therefore decided not to use these techniques to fit the neural network model.

CHAPTER
FOUR

RESULTS AND DISCUSSION

4.1 Offline results

4.1.1 Timeseries problem

As offline results we refer to models performance retrieved on Python, before the actual simulation.
Since the aim of this work was to find a suitable ML algorithm capable of completely replacing the output of the robot's bumper sensors, the focus was not only on detecting a collision, but also on predicting its duration.
Before starting to build a ML model it is helpful to focus on the features available to identify which are most important for predicting a collision.

Since Gyroscopes are crucial sensors for stabilizing the robot, a WMR can keep track of its orientation and adjust its movements accordingly to them. They play a significant role in enabling a WMR to move smoothly and respond to changes in its environment.
At this point, we focused mainly on them to see if there is a sort of correlation between the gyroscope and a collision. As we can see from Figure 4.1.1, the entire original dataset was split into two, depending on whether the data were collision related or not.
The importance of this feature can be seen from the different trend for the observations in which the robot crashed. It can be seen that these observations have more pronounced values, a sign that a collision could be a possible consequence of an abrupt change in angular velocity.

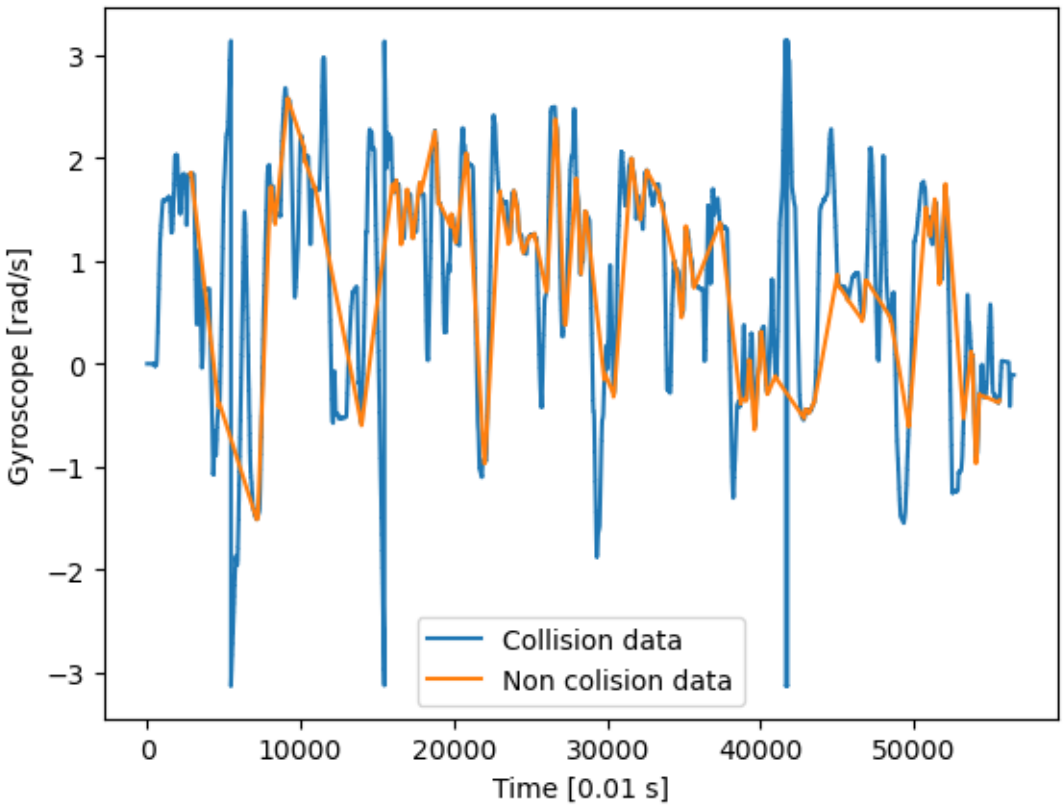


Figure 4.1.1: Gyroscope trends

Furthermore, in order to see if there was any concrete correlation between the features of the moments before the collisions and the collision itself the original dataset was modified to predict only the impact of the robot. As a result, whereas previously the class assumed a value of '1' throughout the entire time of collision, it now only assumes this value at the first instant of the collision. In other words, the data relating to the persistence of the collision after the impact has been eliminated, so that only the impact itself can be predicted in this case.
As a result, the latter is composed as follows:

- "0" if the device follows its path without hitting an obstacle
- "1" if the device has just hit an obstacle, i.e. the first instant of collision

In this way, the fitting algorithm was run for each ML model, in which the target variable is predicted at each time point by the features of previous time points. Therefore, it was decided to run the model fitting algorithm from the current instant until the features from two seconds earlier are considered, each time using these features to predict the response variable of the current instant. This simple idea can be schematised:

$\star$

$$y(t) \xleftarrow{fitting} X(t-k) \qquad \forall k \in 1, ..., 200 \tag{4.1}$$

Where t is the reference time and corresponds to a single row in the original dataset.
As can be seen from Figure 4.1.2, the best score (indicated by a yellow dot) was achieved for features that came from a few instants earlier. More in detail, taking into account the best ML model, which is a Decision tree, the best score to detect a collision is made

with the features that came $K = 2$, instant before than the impact[1], suggesting that it is indeed relevant to consider the previous features at the current time for a correct implementation of the model.
Beyond this point, further increases in k show how the model tends to perform worse and worse, where the worst performance is indicated by the cross cross.
Eventually, this general trends can be summarised in the importance of including instants just before the present ones in the final model.

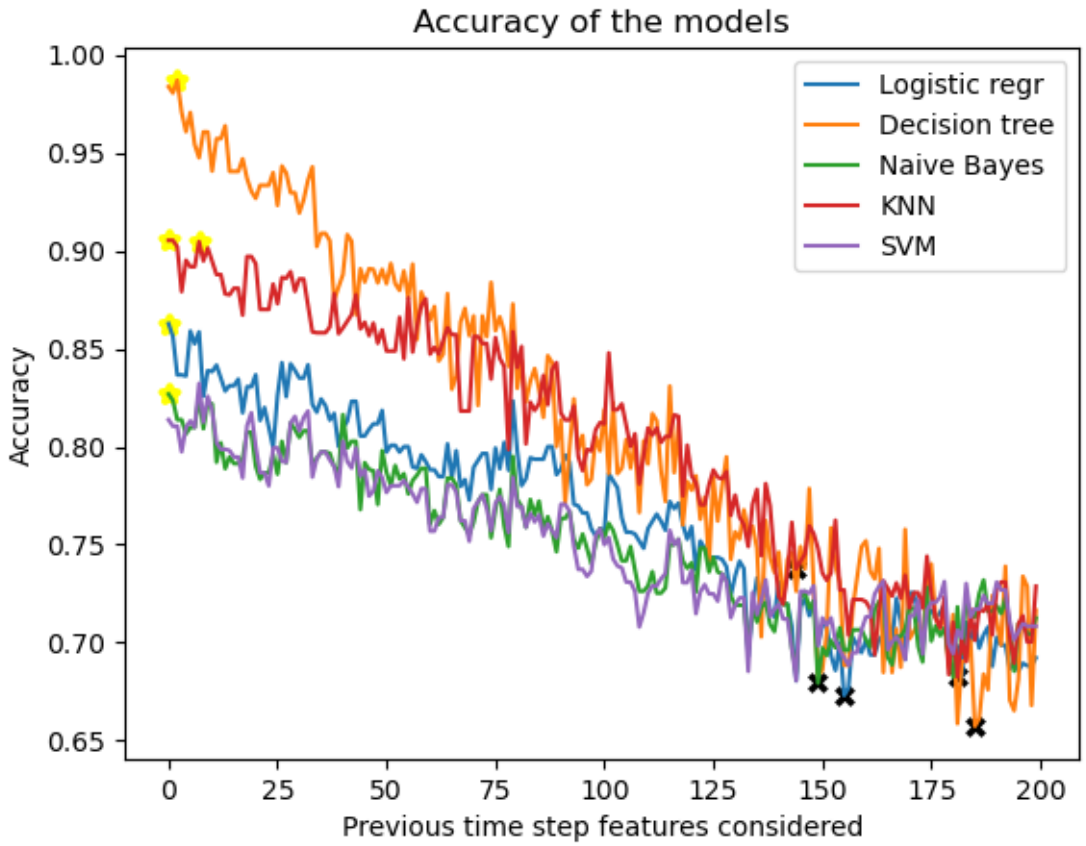


Figure 4.1.2: Models accuracy

A neural network was also run to support this reasoning. From the analysis of Figure 4.1.3, it can be seen that the Loss function decreases to its global minimum when predictions are made with a slightly earlier time (i.e. with a very small k-value). From here, we can see how the loss function grows more and more, and thus how, from k values greater than a certain threshold, the model no longer benefits, but actually gets worse and worse. This consideration is in line with our assumptions, since it is reasonable to think that the features of the moments just before, can be correlated with the present ones.
On the other hand, we also expected that there would be no advantage in considering signals that were too far in the past. Beyond this point, it would be counterproductive and computationally expensive to include additional features from even earlier instants.

[1]Here, the term *instant* refers to fixed-time of simulation, i.e. 0.01 seconds

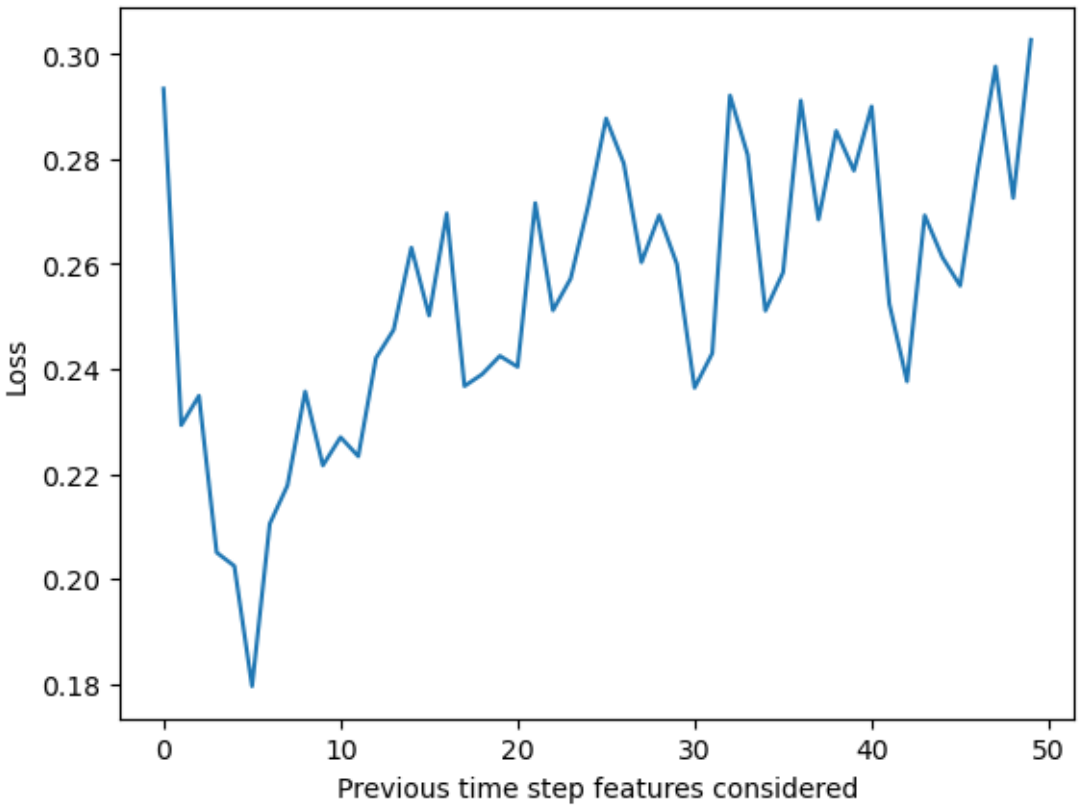


Figure 4.1.3: Binary cross-entropy loss function

In Figure 4.1.3 the ANN was computed starting from predicting $y(t)$ with $X(t)$ and finishing from predicting $y(t)$ with $X(t-50)$, (i.e. 50 iterations of training model)[2].
The loss function was retrieved at the end of the last *epoch*, (10 epochs selected in this case).

4.1.2 Best ML model

The next step was therefore to build an ML model with the logic described in 3.6.[3]
The ML methods described above were then run, followed by different configurations of neural networks. To find the best neural network, random parametric values of *hidden layers* and *neurons* were chosen, identifying the neural network which gives better performance .
On the one hand, it can be seen from Figure 4.1.4 that all the generated models improve their score as the value of k considered increases, until it becomes largely irrelevant to consider features from earlier instants. By this we mean, as before, that there is a certain value of k that makes the model better than the standard one at **3.5**. Otherwise, if we cross this k value threshold, these models will not be able to improve anymore, resulting only in more computation time for the algorithm.
In Figure 4.1.4, the x-axis is described, once again, by the number of features taken into account for each run, i.e. $X(t) + \sum_{k=1}^{K} X(t-k)$.

[2]In this scenario $y(t) \xleftarrow{fitting} X(t-k) \qquad \forall k \in 1, ..., 50$

[3]We have referred to it as a neural network, although in this case it would be more accurate to refer to it as deep learning rather than ML.

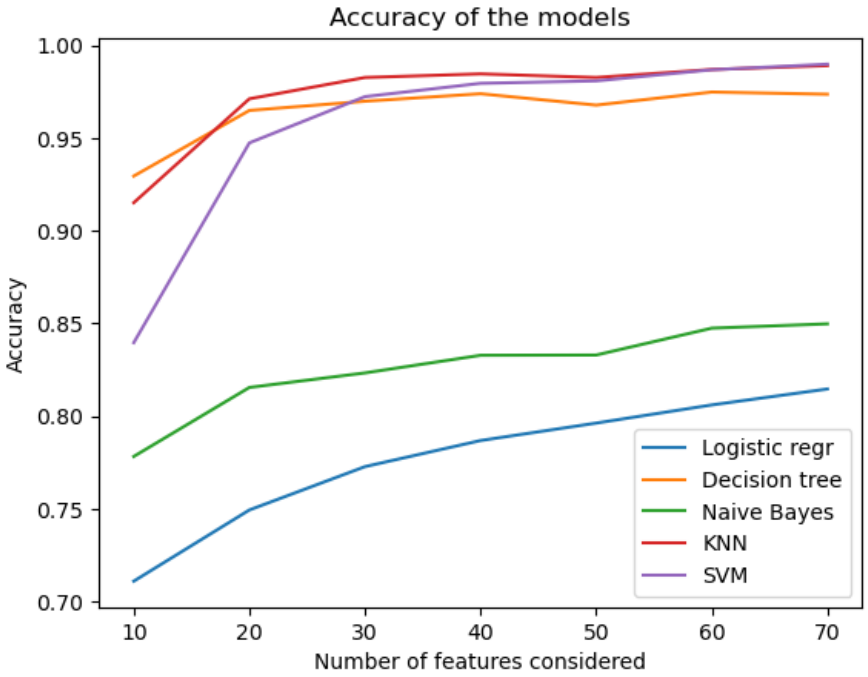


Figure 4.1.4: Models accuracy for different ML models

On the other hand, in order to evaluate the quality of the neural network in question, we decided to analyze both its accuracy loss function during the validation set, as we can see from Figures 4.1.5, 4.1.6. These metrics have been computed on the validation set in order to assess how well an ANN is able to generalize to unseen data, allowing the model itself to detect overfitting and ensuring the model's ability to perform on new data.

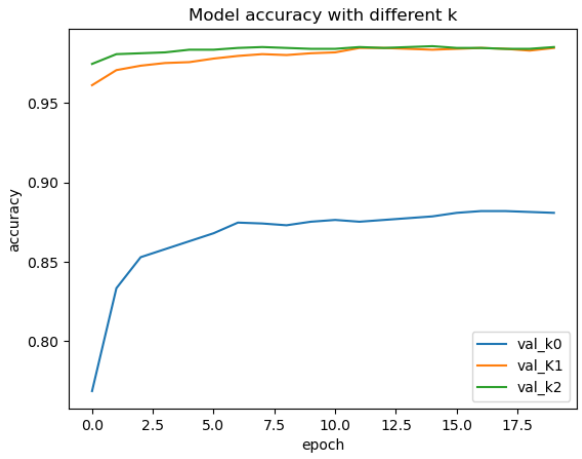


Figure 4.1.5: Model accuracy on validation set

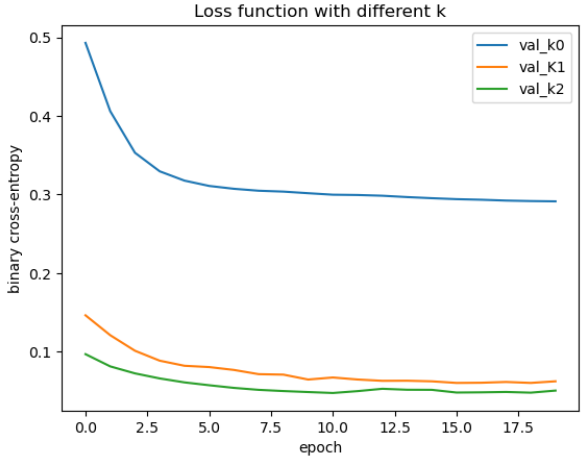


Figure 4.1.6: Binary cross-entropy loss function on validation set

Where:

- **val_k0** refers to the validation set about the ANN standard model described in(3.5)
- **val_k1** refers to the validation set about the ANN that predicts $y(t)$ taking into account the features $X(t) + X(t-1)$
- **val_k2** refers to the validation set about the ANN that predicts $y(t)$ taking into account the features $X(t) + X(t-1) + X(t-2)$

Finally, the best model in Figure 4.1.4 turned out to be the KNN with a performance of 98.2%, present at K=3[4].
In contrast, the neural network achieved a performance of 98.7% at K=1 and 99.3% at K=2, as shown in the confusion matrix in Figure 4.1.7.

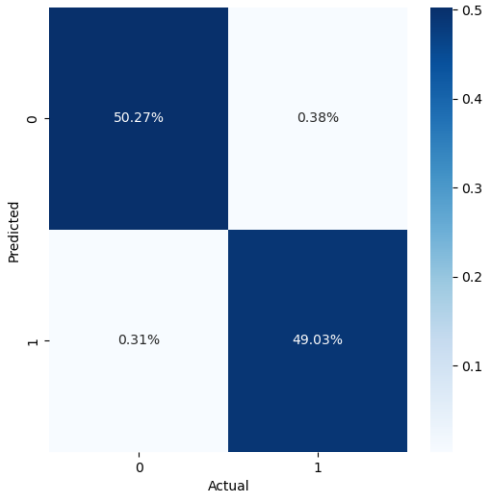


Figure 4.1.7: Confusion matrix for the ANN with K=2

4.2 Online results

From the above results, we decided to consider both neural network models with K=1 and K=2, i.e:

⋆
$$y(t) \xleftarrow{fitting} X(t) + X(t-1) \tag{4.2}$$

⋆
$$y(t) \xleftarrow{fitting} X(t) + X(t-1) + X(t-2) \tag{4.3}$$

From this point, it was finally possible to make the Digital twin model of the Qbot capable of hosting the two models that had been built.
Furthermore, we have noticed that whenever we run the experiment, there is an initial bias which takes a few seconds for the neural network model to function properly.
As described in the previous chapter, given the last activation function of the model,

[4]Actually, the SVM model also exhibited the same performance as the KNN, but was not taken into account as it had it at K=7

3.4, a threshold function was then created to obtain an output corresponding to a single classification value. The identification of an optimal threshold was done in a completely empirical way, by evaluating a correct trade-off of the model's performance between the number of *missed detection* and the number of *false alarms*.
Since, as we saw at the beginning, this model was initially a little too sensitive to abrupt speed changes, resulting in a high false alarm value, the best identified threshold function can be described as follows:

$$\mathbf{y(t)} = \begin{cases} 1 & \text{if } S(y)_{i=1} >= 0.7 \\ 0 & \text{otherwise} \end{cases} \tag{4.4}$$

As we wanted to remain more cautious in the probability of predicting false alarms, this value is slightly higher than the theoretically one expected (0.5), thus achieving a better trade-off.

Once we have run the final simulation, which lasted for 2 minutes, we can first of all see the Figures 4.2.1, 4.2.2 which represent the output of the robot's bumper sensors on one hand, and the output of the neural network model on the other, (first considering K=1, then K=2).
As mentioned above, both models aim to detect not only the robot's collisions but also their duration. In essence, this model tries to emulate the behaviour of the bumper sensor as closely as possible.

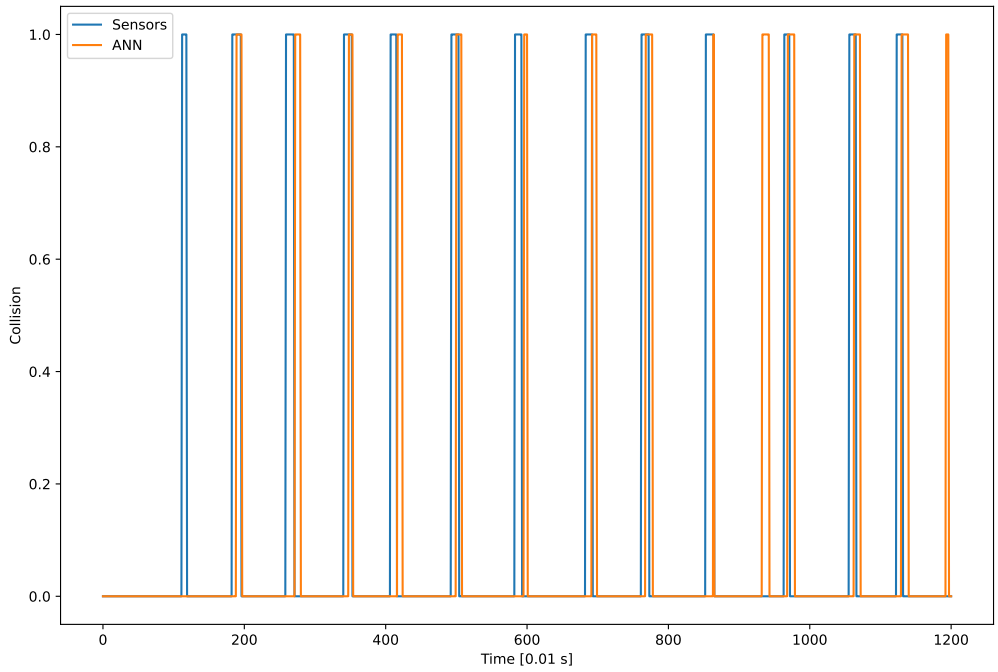


Figure 4.2.1: Real-time collision detection for ANN with K=1

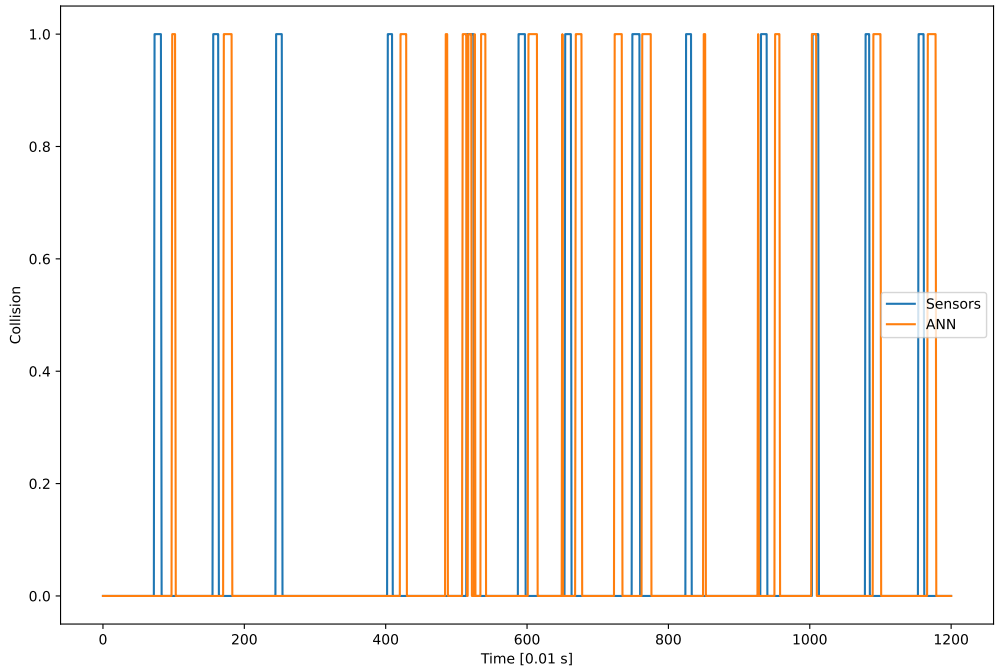


Figure 4.2.2: Real-time collision detection for ANN with K=2

Finally, it was possible to run the simulation to verify the goodness of the model itself. We then calculated these numerical indices of the real-time model's performance:

1. Number of False alarms and Missed detection
2. Average Delay in collision detection
3. Average error in collision duration

Regarding the first two points in Table 4.2.1, the number of false alarms is the number of predicted collisions that did not occur during the simulation, while the number of missed detection is the number of collisions that occurred but were not predicted by the neural network model. These metrics can be described by:

$$False\ Alarms = \frac{Number\ of\ false\ collision\ detection}{Number\ of\ actual\ collisions} \tag{4.5}$$

$$Missed\ Detection = \frac{Number\ of\ missed\ detection}{Number\ of\ actual\ collisions} \tag{4.6}$$

However, these two indicators are very sensitive to changes of even a few moments in the prediction. For example, if two different collisions predictions occur within a few moments of each other for the same collision detected by the sensor, this indicator will count them as separate.

The second point in Table 4.2.1 was computed by taking into account only the difference between the first instant of the collision perceived by the sensor and the first instant of

the collision detected by the neural network (i.e. the moment of impact).
In this way, we did not consider the duration of the collision, but we focused mainly on detecting the exact moment of impact [5].

Instead, the last point refers to the average difference between the **duration** of the collision predicted by the neural network and the duration of the collision detected by the sensor during the simulation. In this case, we did not consider the delay to detect a collision, but only its duration.
We can therefore summarise the results obtained in these real-time performance indicators in Table 4.2.1, [6]:

Performance indicator	$ANN_{K=1}$	$ANN_{K=2}$
False alarms	0.153	0.46
Missed detection	0.07	0.07
Average Delay in collision detection	0.325 s	0.340 s
Average error in collision duration	0.148 s	0.098 s

Table 4.2.1: Performance real-time indicators

As we can see from Table 4.2.1, although the $ANN_{K=2}$ model is more accurate in terms of collision duration, it has a serious weakness in terms of the number of false alarms. This model is therefore too sensitive and, as can be seen in Figure 4.2.2, predicts more collisions than have actually occurred, resulting in a lower level of reliability.

On the other hand, the $ANN_{K=1}$ model manages to perform very well in a real-time simulation for all the indicators considered, being very efficient both in the timely detection of collisions, with an average delay of only 0.3 s, and in the correct estimation of their duration.
As shown in Figure 4.2.1, this model manages to capture each real collision detected by the sensor more accurately than the other neural network, ($ANN_{K=2}$).

Finally, for all the reasons mentioned above, we believe that the $ANN_{K=1}$ model is the best alternative to support or replace the bumper sensor of the robot in case of malfunction or failure.

[5] However, we assume that the average delay in detecting the robot's impact is not only influenced by the goodness of the model, but also by the computation time taken by the Simulink model of the robot's Digital twin itself

[6] The number of collisions that actually occurred was the same in both final simulations, (13)

CHAPTER

FIVE

CONCLUSIONS

5.1 Conclusion

When traditional bumper sensors fail or require maintenance, the robot may need to be withdrawn from service, causing downtime. As a result, ML models can allow the system to continue operating in the event of sensor failure, minimizing disruptions to robot operations.
As these techniques can be applied to multiple WMRs, once a model has been trained and validated, it can be used across a fleet of similar WMRs, potentially reducing maintenance costs on a wider scale. Moreover, in a possible scenario where the bumper sensor shows signs of degradation, the signals from the sensor itself could be replaced by those from the developed neural network model. In this way, a user would still be able to send and receive collision signals from the robot even if the bumper material is faulty and no longer able to send or receive signals correctly, without noticing the difference.
In addition, our research has demonstrated the promising benefits of integrating ML techniques into the collision detection system of WMRs. By replacing traditional bumper sensors with ML algorithms, we have achieved improved adaptability and reliability. The ML-based approach not only improves the robustness and fault tolerance of the WMR, but also minimizes the impact of deterioration on the physical components, resulting in long-term cost savings.
Summarising, the development and implementation of mechatronic systems incorporating ML models has been found to reduce the reliance on preventive maintenance, thereby reducing downtime due to sensor failure or replacement and improving the reliability and durability of WMRs in industrial applications. In addition, the flexibility to continuously update and evolve ML algorithms allows easy integration of enhancements over time without the need to physically change hardware components.

5.2 Model limitation

As we have seen, the neural network model developed for this problem performed particularly well. However, it is important to highlight a critical aspect of our research: the intrinsic variability of the simulations. Each simulation presented different conditions, mainly because the robot was controlled by a joystick rather than following a pre-defined standard path.

This meant that we had to manage slightly different conditions in each simulation, with different simulation parameters changing each time, such as the number of robot collisions.
As a consequence, these limitations required the collection of a large dataset from which to train the ML models, representing all the possible behaviours that the robot itself might exhibit in the future. Consequently, the quality of the ANN model was not affected by this restrictions.
Nevertheless, in order to better compare different experiments, the final performance metrics of the real-time model suffered from these limitations, as they should have been based on a default number of collisions and possible movements of the robot, which actually varied for each final simulation.

5.3 Future work

In the course of this thesis, we have investigated the challenges and advances in the field of real-time collision detection for wheeled mobile robots.
While our work has achieved encouraging results, several opportunities for future research remain to be explored.In this section, we will consider some potential research directions that could further enhance the field of collision detection and enable wheeled mobile robots to operate more effectively in complex environments.

5.3.1 Explainable neural network

Interpretability and explainability are two key factors that are extremely important for applications such as autonomous cars and robots, where it is useful to understand why a car has taken an action, especially if that car or robot is involved in an accident, [53].
Traditional neural networks excel at learning complex patterns and making predictions, including collision detection. However, they often operate as 'black boxes', giving users and researchers limited insight about the reason why certain predictions are made.
A recent approach, called *Explainable Neural Networks*, (XNN), is based on existing deep learning models, such as neural networks, but offers a new deep learning architecture that combines reasoning and learning in synergy. The proposed approach can be described as a feed-forward neural network with an incremental learning algorithm that autonomously evolves its structure to reflect possible dynamic changes, [53].
These models are designed to bridge the gap between model performance and human comprehension. By incorporating XNNs into a collision detection framework, there is an opportunity to provide not only accurate predictions, but also transparent explanations for those predictions, [54].
In dynamic environments where safety and reliability are critical, this opacity can be a cause for concern, and it's essential to understand why a robot might make certain decisions.
Eventually the use of XNNs in real-time simulation dynamics promises to increase the overall safety, reliability and usability of WMR systems, in addition to improving the accuracy of collision detection. This symbiotic relationship between XNNs and real-time collision detection corresponds to the changing robotics environment, where transparency and reliability are key principles in the search for safer and more effective autonomous systems.

5.3.2 Sensor fusion

In this scenario, using a *sensor fusion* technique would be another possible way to achieve high performance in real-time collision detection of a WMR.
Sensor fusion is a technique that combines data from several different sensors to produce a final filtered data that reduces the effects of noise from each sensor used, [55]. In this way it is possible to achieve a more accurate and reliable understanding of the environment than what could be achieved using individual sensors alone.
By enhancing the perception, decision-making and overall accuracy of various systems, this process significantly improves their performance. As a result, sensor fusion is widely used in various fields and it is essential for many AI applications, such as robotics and autonomous driving, [55].
Moreover, in production techniques each industrial robot could represent an hazard zone that changes based on its location and trajectory, a real-time sensor-based approach is becoming increasingly relevant for ensuring the safety of people in close proximity to robots in an industrial workcell.
A possible approach could regard the fusion of data from multiple 3D imaging sensors of different modalities into a volumetric evidence grid and segments the volume into regions corresponding to background, robots, and people. manufacturing practices, [56]. Then, sensor fusion can also be used to provide an improvement in fault tolerance in the event of a malfunction or failure of a sensor. Therefore, the risk of accidents caused by sensor failure is reduced because the system can continue to detect collisions even if one sensor stops working reliably.
Finally, this technology ensures the safety of the WMR, reduces downtime and allows the integration of future sensor technologies, making it a valuable enhancement for collision detection.

REFERENCES

[1] Xinyu Gao et al. "Review of Wheeled Mobile Robots' Navigation Problems and Application Prospects in Agriculture". In: *IEEE Access* 6 (2018), pp. 49248–49268. DOI: 10.1109/ACCESS.2018.2868848.

[2] Rajibul Huq et al. "QBOT: An educational mobile robot controlled in MATLAB Simulink environment". In: *2009 Canadian Conference on Electrical and Computer Engineering*. 2009, pp. 350–353. DOI: 10.1109/CCECE.2009.5090152.

[3] Quanser. *QUARC Real-Time Control Software*. 2020. URL: https://www.quanser.com/products/quarc-real-time-control-software/#overview.

[4] *Qbot 2e*. Quanser. 2019.

[5] Amir Mosavi and Annamaria R. Varkonyi-Koczy. "Integration of Machine Learning and Optimization for Robot Learning". In: *Recent Global Research and Education: Technological Challenges*. Ed. by Ryszard Jabłoński and Roman Szewczyk. Cham: Springer International Publishing, 2017, pp. 349–355. ISBN: 978-3-319-46490-9.

[6] Hugh F. Durrant-Whyte. "An Autonomous Guided Vehicle for Cargo Handling Applications". In: *The International Journal of Robotics Research* 15.5 (1996), pp. 407–440. DOI: 10.1177/027836499601500501. URL: https://doi.org/10.1177/027836499601500501.

[7] Anthony Le et al. "Distributed Vision-Based Target Tracking Control Using Multiple Mobile Robots". In: *2018 IEEE International Conference on Electro/Information Technology (EIT)*. 2018, pp. 0471–0476. DOI: 10.1109/EIT.2018.8500184.

[8] Marc Vazquez, Mateusz Ardito-Proulx, and Sabiha Wadoo. "Lyapunov Based Trajectory Tracking Dynamic Control for a QBOT-2". In: *2020 IEEE Integrated STEM Education Conference (ISEC)*. 2020, pp. 1–6. DOI: 10.1109/ISEC49744.2020.9397845.

[9] *EXPERIMENT 2: LOCOMOTION AND KINEMATICS*. Quanser. 2019.

[10] Tobias Fauser, Stephen B. H. Bruder, and Aly I. El-Osery. "A comparison of inertial-based navigation algorithms for a low-cost indoor mobile robot". In: *2017 12th International Conference on Computer Science and Education (ICCSE)* (2017), pp. 101–106. URL: https://api.semanticscholar.org/CorpusID:20630797.

[11] Howie Choset et al. *Principles of robot motion: theory, algorithms, and implementations*. MIT press, 2005.

[12] A. Bonarini, M. Matteucci, and M. Restelli. "A kinematic-independent dead-reckoning sensor for indoor mobile robotics". In: *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*. Vol. 4. 2004, 3750–3755 vol.4. DOI: 10.1109/IROS.2004.1389998.

[13] Mike Murray. *HOW ROTARY ENCODERS WORK – ELECTRONICS BASICS*. 2019. URL: https://www.thegeekpub.com/245407/how-rotary-encoders-work-electronics-basics/.

[14] Sebastian Dudzik. "Application of the motion capture system to estimate the accuracy of a wheeled mobile robot localization". In: *Energies* 13.23 (2020), p. 6437.

[15] Tom M Mitchell. "The need for biases in learning generalizations". In: (1980).

[16] Thomas G Dietterich and Eun Bae Kong. "Machine learning bias, statistical bias, and statistical variance of decision tree algorithms". In: (1995).

[17] Sethu Vijayakumar. "Bias–variance tradeof". In: (2007).

[18] Gareth James et al. *An Introduction to Statistical Learning: with Applications in R*. Springer, 2013. URL: https://faculty.marshall.usc.edu/gareth-james/ISL/.

[19] Leo Breiman. "Some properties of splitting criteria". In: *Machine learning* 24 (1996), pp. 41–47.

[20] Shagufta Tahsildar Chainika Thakar. *Gini Index: Decision Tree, Formula, and Coefficient*. 2022. URL: https://ibkrcampus.com/ibkr-quant-news/gini-index-decision-tree-formula-and-coefficient/.

[21] Jason Brownlee. *Information Gain and Mutual Information for Machine Learning*. 2019. URL: https://machinelearningmastery.com/information-gain-and-mutual-information/.

[22] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[23] J. Ross Quinlan. "Induction of decision trees". In: *Machine learning* 1 (1986), pp. 81–106.

[24] James Lani. *Homoscedasticity*. 2013. URL: https://www.statisticssolutions.com/free-resources/directory-of-statistical-analyses/homoscedasticity/.

[25] Risma Febrianti, Yekti Widyaningsih, and Saskya Soemartojo. "The parameter estimation of logistic regression with maximum likelihood method and score function modification". In: *Journal of Physics: Conference Series* 1725 (Jan. 2021), p. 012014. DOI: 10.1088/1742-6596/1725/1/012014.

[26] Pia Veldt Larsen. "In All Likelihood: Statistical Modelling and Inference Using Likelihood". In: *Journal of the Royal Statistical Society Series D: The Statistician* 52.3 (Aug. 2003), pp. 416–417. ISSN: 2515-7884. DOI: 10.1111/1467-9884.00369_20. eprint: https://academic.oup.com/jrsssd/article-pdf/52/3/416/49946098/jrsssd_52_3_416a.pdf. URL: https://doi.org/10.1111/1467-9884.00369%5C_20.

[27] "PSEUDO-R 2 IN LOGISTIC REGRESSION MODEL". In: *Statistica Sinica* 16.3 (2006), pp. 847–860. ISSN: 10170405, 19968507. URL: http://www.jstor.org/stable/24307577 (visited on 09/19/2023).

[28] Paul D. Allison. "Measures of Fit for Logistic Regression". In: 2014. URL: https://api.semanticscholar.org/CorpusID:13909621.

[29] *Statistics: a Bayesian perspective*. Vol. 4. 1996.

[30] Daniel Berrar. "Bayes' Theorem and Naive Bayes Classifier". In: Jan. 2018. ISBN: 9780128096338. DOI: 10.1016/B978-0-12-809633-8.20473-1.

[31] Heibe Frank Ian H.Witten. *Data Mining Practical Machine Learning Tools and Techniques, 2 edition*. Elsevier, 2011.

[32] Department of Computer Science / Finance K. Ming Leung POLYTECHNIC UNIVERSITY and Risk Engineering. *Naive Bayesian Classifier*. 2007. URL: https://cse.engineering.nyu.edu/~mleung/FRE7851/f07/naiveBayesianClassifier.pdf.

[33] Vikramaditya Jakkula. "Tutorial on support vector machine (svm)". In: *School of EECS, Washington State University* 37.2.5 (2006), p. 3.

[34] Anna-Lena Popkes. *Extensive Guide to Support Vector Machines*. 2021. URL: https://www.inovex.de/de/blog/support-vector-machines-guide/.

[35] GRID inc. *Inside the activation function*. 2018. URL: https://www.renom.jp/notebooks/tutorial/basic_algorithm/activation/notebook.html.

[36] Marco Lippi. *Machine Learning*. 2021.

[37] Trong-Ton Pham. "MODELE DE GRAPHE ET MODELE DE LANGUE POUR LA RECONNAISSANCE DE SCENES VISUELLES". In: (Dec. 2010).

[38] GREGORY G. ENAS and SUNG C. CHOI. "CHOICE OF THE SMOOTHING PARAMETER AND EFFICIENCY OF k-NEAREST NEIGHBOR CLASSIFICATION". In: *Statistical Methods of Discrimination and Classification*. Ed. by SUNG C. CHOI. Pergamon, 1986, pp. 235–244. ISBN: 978-0-08-034000-5. DOI: https://doi.org/10.1016/B978-0-08-034000-5.50011-3. URL: https://www.sciencedirect.com/science/article/pii/B9780080340005500113.

[39] Gongde Guo et al. "KNN Model-Based Approach in Classification". In: *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*. Ed. by Robert Meersman, Zahir Tari, and Douglas C. Schmidt. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 986–996. ISBN: 978-3-540-39964-3.

[40] Ravil Muhamedyev. "Machine learning methods: An overview". In: *Computer modelling & new technologies* 19.6 (2015), pp. 14–29.

[41] Stuart J Russell. *Artificial intelligence a modern approach*. Pearson Education, Inc., 2010.

[42] Waleed Yamany et al. "A New Multi-layer Perceptrons Trainer Based on Ant Lion Optimization Algorithm". In: Sept. 2015, pp. 40–45. DOI: 10.1109/ISI.2015.9.

[43] Jason Brownlee. *How to Control the Stability of Training Neural Networks With the Batch Size*. 2020. URL: https://machinelearningmastery.com/how-to-control-the-speed-and-stability-of-training-neural-networks-with-gradient-descent-batch-size/.

[44] Sweta. *Batch , Mini Batch and Stochastic gradient descent*. 2020. URL: https://sweta-nit.medium.com/batch-mini-batch-and-stochastic-gradient-descent-e9bc4cacd461.

[45] Code: Robotics. *Accelerometer*. 2019. URL: https://docs.idew.org/code-robotics/references/physical-inputs/accelerometer.

[46] Mike Folk et al. "An overview of the HDF5 technology suite and its applications". In: *Proceedings of the EDBT/ICDT 2011 workshop on array databases*. 2011, pp. 36–47.

[47] Chenguang Wan et al. "A Robust and Fast Data Management System for Machine-Learning Research of Tokamaks". In: *IEEE Transactions on Plasma Science* 50.12 (2022), pp. 4980–4986. DOI: 10.1109/TPS.2022.3223732.

[48] *Open Neural Network Exchange The open standard for machine learning interoperability*. URL: https://onnx.ai/.

[49] Quanser. *Stream Client*. 2019. URL: https://docs.quanser.com/quarc/documentation/quarc_communications_basic.html.

[50] Vishal Yathish. *Loss Functions and Their Use In Neural Networks*. 2022. URL: https://towardsdatascience.com/loss-functions-and-their-use-in-neural-networks-a470e703f1e9.

[51] Kiprono Elijah Koech. *Softmax Activation Function — How It Actually Works*. 2020. URL: https://towardsdatascience.com/softmax-activation-function-how-it-actually-works-d292d335bd78.

[52] Jason Bronwlee. *Softmax Activation Function with Python*. 2020. URL: https://machinelearningmastery.com/softmax-activation-function-with-python/.

[53] Plamen Angelov and Eduardo Soares. "Towards explainable deep neural networks (xDNN)". In: *Neural Networks* 130 (2020), pp. 185–194. ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2020.07.010. URL: https://www.sciencedirect.com/science/article/pii/S0893608020302513.

[54] Fatai Sado et al. "Explainable Goal-driven Agents and Robots-A Comprehensive Review". In: *ACM Computing Surveys* 55.10 (2023), pp. 1–41.

[55] Biswaindu Parida. *Sensor Fusion: The Ultimate Guide to Combining Data for Enhanced Perception and Decision-Making*. 2023. URL: https://www.wevolver.com/article/what-is-sensor-fusion-everything-you-need-to-know.

[56] Paul Rybski et al. "Sensor fusion for human safety in industrial workcells". In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2012, pp. 3612–3619. DOI: 10.1109/IROS.2012.6386034.

APPENDICES

```
1
2  #MAIN ALGORITHM FOR THE NEURAL NETWORK
3  import pandas as pd
4  import numpy as np
5  from matplotlib import pyplot as plt
6
7  from sklearn.model_selection import train_test_split
8  from sklearn.preprocessing import StandardScaler, MinMaxScaler
9  from sklearn.metrics import confusion_matrix
10 from seaborn import heatmap
11 import seaborn as sbn
12 from time import process_time
13
14 #For other ML models
15 from sklearn.linear_model import LogisticRegression
16 from sklearn.neighbors import KNeighborsClassifier
17 from sklearn.tree import DecisionTreeClassifier
18 from sklearn.naive_bayes import GaussianNB
19 from sklearn.svm import SVC
20
21 #For Neural network model
22 from keras.models import Sequential
23 from keras.optimizers import Adam
24 from keras.metrics import Recall
25 from keras.metrics import BinaryAccuracy
26 from keras.layers import Dense
27 from keras.callbacks import EarlyStopping
28
29
30
31 #Dataset retrieved from 20 minute Qbot simulation
32 df = pd.read_excel('Qbot_Simulation.xlsx')
33
34 #for plots
35 it = []
36 Accuracy = []
37 ValAccuracy = []
38 Loss = []
39 ValLoss = []
```

```
40
41 #inizialization
42 Xt = pd.DataFrame() #starting features dataframe
43 X = df.iloc[:,:-1]
44
45
46 #Neural network MODEL that takes into account also the "k"
   features from the previous instant "t"
47 #y(t) is predicted by X(t)+X(t-1)+...+X(t-k)
48 #Since in real-time performance on Simulink we do not have
   standardize signals, here the features are not standardized
49
50 y = df['BUMP']  #target variable which refers to bumper sensors
51 X = df.drop(['BUMP'], axis=1)
52
53
54 k = 4 #parameter
55 for i in range(0, k):
56     Xt = pd.concat([Xt, X.shift(i)], axis=1)
57     y = df['BUMP']
58     y = pd.get_dummies(y) #needed for this type of model
59
60     df_unito = pd.concat([Xt, y], axis=1).dropna()
61
62     it.append(10*i) #for future plot
63
64     #Splitting
65     Xtrain, Xtest, ytrain, ytest = train_test_split(df_unito.iloc
          [:,:-2], df_unito.iloc[:, -2:], test_size = 0.3,
          random_state=42)
66
67
68     model = Sequential()
69     model.add(Dense(units = 16, input_dim=Xtrain.shape[1],
          activation='relu'))
70     model.add(Dense(units = 32, activation='relu'))
71     model.add(Dense(units = 2, activation='softmax'))
72
73     model.compile(loss='binary_crossentropy', metrics = ['acc'],
          optimizer = Adam(learning_rate=0.001) )
74     early_stop = EarlyStopping(monitor='loss', patience=5, verbose
          =1)
75
76     #Fitting
77     history = model.fit(Xtrain, ytrain, validation_split=0.1,
          epochs=20, batch_size=6, verbose=0,callbacks=[early_stop],
          shuffle=False)
78
79     #For plotting
80     Accuracy.append(history.history['acc'])  #sono: n_epoch for
          each loop
```

```
    ValAccuracy.append(history.history['val_acc'])
    Loss.append(history.history['loss'])
    ValLoss.append(history.history['val_loss'])


ypred_test =  model.predict(Xtest)
ypred_test = np.round(ypred_test)
ypred_test = pd.DataFrame(ypred_test)

#Retrasform from one-hot encoding format to one column with '0'
   and '1' values:
ypred_test = ypred_test.idxmax(axis=1)
ytest = ytest.idxmax(axis=1)

#Confusion matrix
cm = confusion_matrix(ypred_test, ytest)
plt.figure(figsize = (6,6))
plt.xlabel
heatmap(cm/sum(sum(cm)), annot = True, fmt = '.2%', cmap = 'Blues'
   )
plt.xlabel('Actual')
plt.ylabel('Predicted')
plt.show()

#To make the Neural network readable on Simulink, (.h5 format)
model.save('modelANN_k1.h5')


#To import the neural network in Simulink (MATLAB SCRIPT)
file = 'modelANN_K1.h5';
net = importKerasNetwork(file);
save('modelANN_k1.mat', 'net')
#Then, the following neural network will go into the 'Predict'
   block of 'Deep Learning toolbox'
```