



MARCOS MIGUEL TEIXEIRA PEREIRA MATEUS

Licenciado em Ciências da Engenharia Electrotécnica e de
Computadores

END-TO-END PERFORMANCE ANALYSIS OF A RESOURCE ALLOCATION SERVICE

MESTRADO EM ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES

Universidade NOVA de Lisboa
Fevereiro, 2022

END-TO-END PERFORMANCE ANALYSIS OF A RESOURCE ALLOCATION SERVICE

MARCOS MIGUEL TEIXEIRA PEREIRA MATEUS

Licenciado em Ciências da Engenharia Electrotécnica e de Computadores

Orientador: Luís Bernardo
Professor Associado, Universidade Nova de Lisboa

Coorientadora: Daniela Oliveira
Senior System Developer, Skyline Communications

Júri:

Presidente: Anabela Monteiro Gonçalves Pronto
Professora Auxiliar, Universidade Nova de Lisboa

Arguente: Rodolfo Alexandre Duarte Oliveira
Professor Associado, Universidade Nova de Lisboa

Orientador: Luís Bernardo
Professor Associado, Universidade Nova de Lisboa

End-to-end Performance Analysis of a Resource Allocation Service

Copyright © Marcos Miguel Teixeira Pereira Mateus, Faculdade de Ciências e Tecnologia, Universidade NOVA de Lisboa.

A Faculdade de Ciências e Tecnologia e a Universidade NOVA de Lisboa têm o direito, perpétuo e sem limites geográficos, de arquivar e publicar esta dissertação através de exemplares impressos reproduzidos em papel ou de forma digital, ou por qualquer outro meio conhecido ou que venha a ser inventado, e de a divulgar através de repositórios científicos e de admitir a sua cópia e distribuição com objetivos educacionais ou de investigação, não comerciais, desde que seja dado crédito ao autor e editor.

ACKNOWLEDGEMENTS

This work would not have been possible without the collaboration and support of many people special to me.

First, I would like to thank my advisor, professor Luís Bernardo, for the support and precious advice given throughout this course and especially, in this dissertation, for all the patience and availability shown. Thank you for being an excellent advisor.

A special thanks has to be given to my excellent co-advisor, Daniela Oliveira, for the huge help given throughout this dissertation, the feedback provided, the sharing of her expert knowledge, and more importantly, for all the patience. I would also like to thank the members of Skyline Communications's System Developer team, for the help and for the teachings that enabled me to complete this dissertation.

Then, I would like to thank my friends for being supportive and being with me in all moments of my life, and also thanks to all my colleagues for providing me with great years of college.

Last but not less important, I want to thank my family that made so many sacrifices to raise me into the man that I am today. Without them, I would not be where I am now.

*“Education never ends, Watson. It is a series of lessons, with
the greatest for the last.” (Sir Arthur Conan Doyle)*

ABSTRACT

This dissertation is centered around the monitoring and control platform of the company *Skyline Communications*, the DataMiner. This platform has a specific module called SRM (Service and Resource Management). One of SRM's many features is the capacity to make an advance reservation (booking) of resources of the client's network. When a booking is created, there is a time interval/delay between the moment that a booking is requested and the moment that all necessary configurations for this booking actually start to be made at the Resource level. This delay is called SyncTime.

The SyncTime is affected by the dynamics of the network (e.g, a sudden increase in the number of bookings made, at a given time). In order to guarantee the maximum possible quality of service to the client and ensure that the network dynamics will have minimal impact in the real-time delivery of the desired content, the SRM module must be able to estimate if a booking can be done in an acceptable SyncTime value. Given this problem, the main goal of this dissertation is to develop a machine learning based estimation/classification module that is capable of, based on the temporal state of the SRM module, make a prediction or classification of the SyncTime.

Two approaches were considered: Classify the SyncTime based on classes or estimate the value of it. In order to test both approaches, we implemented several traditional machine learning methods as well as, several neural networks. Both approaches were tested using a dataset collected from a DataMiner cluster composed of three DataMiner agents using software developed in this dissertation. In order to collect the dataset, we considered several setups that captured the cluster in different network conditions.

By comparing both approaches, the results suggested that classifying the predicted SyncTime using a classification model and classifying the predicted SyncTime of a estimation model are both equally good options. Furthermore, based on the results of all implementations, a prototype application was also developed. This application was fully developed in Python and it uses a Multilayer Perceptron in order to do the classification of the SyncTime of a booking, based on several inputs given by the user.

Keywords: DataMiner, Data Mining, Classification, Regression, Machine Learning

RESUMO

Esta dissertação centra-se na plataforma de monitorização e controlo da empresa *Skyline Communications*, o *DataMiner*. Esta plataforma possui um módulo específico denominado de SRM (*Service and Resource Management*). Uma das características do SRM é a capacidade de fazer uma reserva antecipada (*booking*) dos recursos da rede de um cliente. Quando uma reserva é criada, existe um intervalo de tempo/atraso entre o momento em que uma reserva é solicitada e o momento em que todas as configurações necessárias para a mesma realmente começam a ser feitas ao nível de Recurso do *DataMiner*. Este atraso é chamado de SyncTime.

O SyncTime é afetado pela dinâmica da rede (por exemplo, um aumento repentino no número de reservas feitas num determinado momento). De forma a assegurar a máxima qualidade de serviço possível ao cliente e garantir que a dinâmica da rede tenha o mínimo impacto na entrega em tempo real do conteúdo desejado, o módulo SRM deve ser capaz de estimar se uma reserva pode ser feita com um valor de SyncTime aceitável. Diante deste problema, o principal objetivo desta dissertação é desenvolver um módulo de regressão/classificação baseado em aprendizagem automática (*machine learning*) que seja capaz de fazer uma previsão do valor do SyncTime ou classificação do mesmo, com base no estado temporal do módulo SRM.

Duas abordagens foram consideradas: Classificar o SyncTime com base em classes ou estimar o seu valor. Para testar as mesmas, implementou-se vários métodos tradicionais de *machine learning*, bem como várias redes neuronais. Ambas as abordagens foram testadas utilizando um conjunto de dados recolhido de um *cluster* composto por três agentes *DataMiner* usando *software* desenvolvido nesta dissertação. Para a recolha dos dados, considerou-se várias configurações que capturam o *cluster* em diferentes condições.

Ao comparar ambas as abordagens, os resultados sugerem que classificar o SyncTime usando um modelo de classificação e classificar o valor do SyncTime previsto por um modelo de regressão são ambas boas opções. Com base nos resultados obtidos, foi criada ainda uma aplicação protótipo. Esta foi totalmente desenvolvida em *Python* e utiliza um *Multilayer Perceptron* para realizar a classificação do SyncTime de uma reserva, a partir dos dados introduzidos pelo utilizador.

Palavras-chave: *DataMiner*, *Data Mining*, Classificação, Regressão, *Machine Learning*

CONTENTS

| | |
|--|-------------|
| List of Figures | xi |
| List of Tables | xiii |
| 1 Introduction | 1 |
| 1.1 Context | 1 |
| 1.2 Objectives | 2 |
| 1.3 Contributions | 2 |
| 1.4 Outline | 2 |
| 2 State of Art | 4 |
| 2.1 Introduction | 4 |
| 2.2 Classification and Regression Problems | 4 |
| 2.2.1 Classification | 4 |
| 2.2.2 Regression | 5 |
| 2.3 Machine Learning Taxonomy | 5 |
| 2.3.1 Supervised Learning | 6 |
| 2.3.2 Unsupervised Learning | 6 |
| 2.3.3 Semi-supervised Learning | 6 |
| 2.4 Principal Components Analysis (PCA) | 7 |
| 2.4.1 Implementation Examples | 8 |
| 2.5 Random forest (RF) | 8 |
| 2.5.1 Implementation Examples | 10 |
| 2.6 Support Vector Machine (SVM) | 11 |
| 2.6.1 Implementation Examples | 12 |
| 2.7 Multilayer Perceptron (MLP) | 13 |
| 2.7.1 Implementation Examples | 15 |
| 2.8 Autoencoders (AE) | 16 |
| 2.8.1 Autoencoder Variants | 18 |
| 2.8.2 Implementation Examples | 18 |
| 2.9 Long Short Term Memory (LSTM) | 19 |
| 2.9.1 Implementation Examples | 20 |
| 2.10 Hybrid ML models | 22 |

| | | |
|----------|--|-----------|
| 2.10.1 | Implementation Examples | 22 |
| 2.11 | ML Methods Evaluation | 24 |
| 2.11.1 | Cross-Validation | 25 |
| 2.11.2 | Confusion Matrix | 25 |
| 2.11.3 | Evaluation Metrics: Classification | 27 |
| 2.11.4 | Evaluation Metrics: Regression | 29 |
| 3 | Dataset Acquisition | 30 |
| 3.1 | Dataset Description | 30 |
| 3.2 | Data Collection Process | 32 |
| 3.2.1 | Booking & Cluster State Information | 32 |
| 3.2.2 | Setups Definition | 34 |
| 3.2.3 | Dataset Setups | 35 |
| 3.3 | Data Aggregation | 36 |
| 4 | Data Analysis | 41 |
| 4.1 | Cumulative Density Function (CDF) | 41 |
| 4.2 | t-SNE: t-distributed Stochastic Neighbor Embedding | 43 |
| 4.3 | Correlation Matrix | 45 |
| 5 | Classification Approach: Implementation | 49 |
| 5.1 | Data Pre-Processing | 49 |
| 5.2 | SVM Implementation | 50 |
| 5.3 | RF Implementation | 52 |
| 5.4 | MLP Implementation | 53 |
| 6 | Regression Approach: Implementation | 59 |
| 6.1 | Data Pre-processing | 59 |
| 6.2 | MLP Regressor Implementation | 60 |
| 6.2.1 | Initial Test | 60 |
| 6.2.2 | 2 Hidden Layers Test | 62 |
| 6.3 | Hybrid Regressor Implementation | 64 |
| 7 | Discussion of Results & Developed Software | 68 |
| 7.1 | Classification Approach | 68 |
| 7.2 | Regression Approach | 70 |
| 7.3 | Classification vs. Regression Approaches | 71 |
| 7.4 | Developed Software | 73 |
| 8 | Conclusion & Future Work | 77 |
| 8.1 | Conclusion | 77 |
| 8.2 | Future Work | 79 |

CONTENTS

Bibliography

80

LIST OF FIGURES

| | | |
|------|---|----|
| 2.1 | PCA with two principal components | 7 |
| 2.2 | Random Forest Decision Scheme | 9 |
| 2.3 | Feature Randomness | 9 |
| 2.4 | Application of Linear SVM on linearly separable data | 11 |
| 2.5 | Linear SVM with Non-linearly Separable Data | 12 |
| 2.6 | MLP Network | 13 |
| 2.7 | Sigmoid Function Plot | 14 |
| 2.8 | ReLu Activation Function | 14 |
| 2.9 | Linear Activation Function | 15 |
| 2.10 | Structure of a autoencoder with one hidden layer | 17 |
| 2.11 | Deep Autoencoder with L layers | 17 |
| 2.12 | Memory Block on LSTM | 21 |
| 2.13 | Split Process done in 5-Fold Cross Validation | 25 |
| 2.14 | General Structure of a Confusion Matrix | 26 |
| 2.15 | ROC Curve | 28 |
| 3.1 | CPUSTRES Main Interface | 34 |
| 3.2 | Clumsy Main Interface | 35 |
| 3.3 | Concurrent Bookings Example | 40 |
| 4.1 | Dataset CDF plot | 42 |
| 4.2 | Class Distribution in the dataset | 43 |
| 4.3 | 2-Dimensional plot of the dataset | 44 |
| 4.4 | Dataset Classes | 44 |
| 4.5 | Zoom In View of Clusters A and B | 45 |
| 5.1 | Loss plot of Configuration 1 | 55 |
| 5.2 | Loss plot of Configuration 2 | 55 |
| 5.3 | Loss plot of Configuration 3 | 56 |
| 5.4 | Loss plot of Configuration 1 using EarlyStopping | 57 |
| 5.5 | Loss plot of MLP with 1 unit | 58 |
| 6.1 | Loss plot of Configuration 2 MLP Regressor - Initial Test | 61 |

LIST OF FIGURES

| | | |
|------|--|----|
| 6.2 | Predicted vs. Actual SyncTime values for MLP Regressor - Initial Test . . . | 62 |
| 6.3 | Loss plot of Configuration 3 MLP Regressor - 2 Hidden Layers Test | 64 |
| 6.4 | Linear Regression Model | 65 |
| 6.5 | Loss plot of Configuration 3 Hybrid Regressor | 66 |
| 6.6 | Loss plot with Overfitting effect | 67 |
| 7.1 | SVM Confusion Matrix | 69 |
| 7.2 | RF Confusion Matrix | 69 |
| 7.3 | MLP Classifier Confusion Matrix | 69 |
| 7.4 | Predicted vs. Actual SyncTime values | 70 |
| 7.5 | Predicted SyncTime values below 13 s | 71 |
| 7.6 | Predicted SyncTime values below 13 s with class I samples identified by red rectangles | 72 |
| 7.7 | MLP Regressor Confusion Matrix | 73 |
| 7.8 | Hybrid Regressor Confusion Matrix | 73 |
| 7.9 | Application Main GUI | 75 |
| 7.10 | New Prediction Diagram | 75 |

LIST OF TABLES

| | | |
|-----|--|----|
| 3.1 | Dataset Setups Table - Part 1 | 37 |
| 3.2 | Dataset Setups Table - Part 2 | 38 |
| 4.1 | Setups Correlation Table | 47 |
| 5.1 | Grid used in the SVM implementation | 51 |
| 5.2 | Best SVM Hyperparameters | 52 |
| 5.3 | Grid used in the RF implementation | 53 |
| 5.4 | Best RF Hyperparameters | 53 |
| 5.5 | Configurations Tested for the MLP Classifier | 54 |
| 6.1 | Configurations Tested for the MLP Regressor - Initial Test | 61 |
| 6.2 | Initial Test MLP Regressor Results | 61 |
| 6.3 | Configurations Tested in the 2 Hidden Layers Test | 63 |
| 6.4 | 2 Hidden Layers Test Results | 63 |
| 6.5 | Hybrid Regressor Configurations | 65 |
| 6.6 | Hybrid Regressor Results | 66 |
| 7.1 | Classification Approach Best Results | 68 |
| 7.2 | Regression Approach Best Results | 70 |
| 7.3 | Classification vs. Regression Approach Results | 72 |
| 7.4 | Comparison between the MLP Classifier and Hybrid Regressor | 73 |

INTRODUCTION

1.1 Context

This dissertation's challenge is centered around the monitoring and control platform of the company *Skyline Communications*, the DataMiner. This platform has a specific module called SRM (Service and Resource Management). Through this module, DataMiner manages the orchestration of resources, bookings and ensures the delivery of multimedia content in real time to the customers network. DataMiner supports any type of network based on various technologies (from mobile networks to satellite networks).

One of the features available in the SRM module is the capacity to make an advance reservation (booking) of resources of the client's network. When a booking is created, there is a time interval between the moment that a booking is requested and the moment that the necessary configurations for this booking start to be made at the Resource level. In resume, this delay corresponds to the time that the system takes to do all the necessary steps to prepare for a booking. In this dissertation, we name this time interval as SyncTime.

The SyncTime value of a booking is affected by the dynamics of the network. These can be, for example, the consequence of the stress put on the network due to a sudden increase in the number of bookings made, at a given time. In order to guarantee the maximum possible quality of service to the client and ensure that the network dynamics will have minimal impact in the real-time delivery of the desired content, the SRM module must minimize as much as possible the SyncTime value of all bookings.

1.2 Objectives

The first objective of this dissertation is to develop an estimation module that is capable of, based on the temporal state of the SRM module, make a prediction or classification of the SyncTime of a booking. This estimation module can be used alongside the SRM module in order to minimize the SyncTime delay of a booking and ensure real-time delivery of the content requested in said booking.

The second objective of this thesis is to build a portable desktop application that integrates one of the delay estimators/classifiers implemented in this thesis. The main functionality of this application will be to predict, based on a set of input data given by the user, the booking's SyncTime value or in alternative, show a label that classifies the booking's SyncTime value based on classes (e.g, Ideal or Not Ideal).

1.3 Contributions

The main contributions of this thesis are:

- We discover which input parameters had more impact in the SyncTime of a booking.
- We selected the best algorithms that can better deal with the estimation and classification of the booking's SyncTime;
- We developed a prototype application which classifies the SyncTime of a booking based on a set of input parameters. This classification is done by a pre-trained Multilayer Perceptron. The goal of this software is to aid in the assessment and development of future SRM projects deployment.

1.4 Outline

After this introductory chapter comes the State of Art chapter (Chapter 2), which gives a brief introduction to the machine learning concepts and to classification and regression problems. Several ML methods are also described, along with implementation examples that show their utility in different scenarios.

Chapter 3 describes the dataset used in this thesis and the data collection process used to acquire it.

Chapter 4 presents the data analysis done and the discoveries that came from it.

Chapters 5 and 6 show the implementation process of some of the ML algorithms studied in chapter 2.

In chapter 7, we selected the best implementations of the previous two chapters and discuss the results obtained by of each of them.

Lastly, chapter 8 resumes all work done in this thesis and provides directions for future work.

2.1 Introduction

In this chapter, we do a brief introduction to the concept of machine learning and address all the machine learning methods considered to be adequate to deal with the problem of this master's thesis.

Two approaches are considered: The first one is the classification of the total time between the request of a booking and the moment the booking configuration is fully executed at the Resource level based on classes (i.e, acceptable and non-acceptable). The other approach is to estimate the value of this time interval. In other words, this problem can be seen from a classification perspective or regression perspective. A review is presented on methods that can be used for classification or regression problems.

2.2 Classification and Regression Problems

2.2.1 Classification

The main goal in a classification problem is to classify the input dataset using different categories or classes. The task of a classification algorithm (or classifier) is to find a function that divides the input dataset into multiple classes, depending on different characteristics/features present on the dataset. The output of a classifier can be a class label (i.e, "ideal" or "not ideal") or it can be a discrete value. This discrete value is associated with the probability of a sample being part of a certain class.

A classification problem can be further divided in two types, depending on the number of output classes:

- **Binary Classification:** It is a type of classification problem, in which a given dataset is classified into two classes. An example of binary classification problem is the email spam detection.
- **Multiclass Classification:** It is a type of classification problem, in which a given dataset is classified into three or more classes. Examples of multiclass classification problems are face classification and plant species classification.

There are several examples of machine learning algorithms that are used in classification problems. Some common examples are the Support Vector Machine (section 2.6) and Random Forest (section 2.5).

2.2.2 Regression

Unlike classification, the main goal of a regression algorithm is to find a function or a model that predicts a continuous real value based on the input data. In other words, as stated in [8], its main task is to approximate a function f that maps input variable x to a continuous output variable y . This output variable is a real-value variable so it can be an integer variable or floating point variable.

Common regression problems are stock price prediction, weather prediction, etc.

There are several examples of machine learning algorithms that are used in regression problems. Some examples are the Linear regression and the Multilayer Perceptron (section 2.7).

2.3 Machine Learning Taxonomy

According to [31], "machine learning is the idea of teaching computers to predict or classify data without being explicitly programmed for such tasks". Different ways of dealing with multiple problems resulted in a wide variety of different machine learning algorithms.

Machine learning algorithms can be classified into three main groups:

- **Supervised Learning;**
- **Unsupervised Learning;**
- **Semi-supervised Learning.**

In the following subsections, we describe each of these three groups in detail.

2.3.1 Supervised Learning

Supervised learning refers to machine learning algorithms that learn a mapping function that associates a collection of input training data points X_1, X_2, \dots, X_p to the corresponding output data points Y_1, Y_2, \dots, Y_p . Supervised learning algorithms work based on equation 2.1.

$$Y = f(X) \tag{2.1}$$

In this equation, X corresponds to a set of input data points, Y corresponds to a set of output data points and f is the mapping function created by a machine learning algorithm.

Some examples of supervised learning algorithms that are commonly used are Support Vector Machine (SVM) (described in section 2.6), Random Forest (described in section 2.5), Decision Trees, Linear regression, Naive Bayes Classifiers and many others [24].

2.3.2 Unsupervised Learning

Unsupervised Learning is the opposite of supervised learning, where only exists input data but no output variables or labels. None of the data can be pre-classified beforehand so, the machine learning algorithm has to model and classify the input data based on distinct features present on the data.

Some examples of unsupervised learning algorithms that are commonly used are Principal Components Analysis (PCA) (described in section 2.4), One-Class SVM, Isolation Forest, K-means, Autoencoder (described in section 2.8) and many others [45].

2.3.3 Semi-supervised Learning

Semi-supervised learning is a mixture of unsupervised and supervised learning because, of the input data points that exist, only a few are labeled (have a corresponding output data point or label) and, the rest is unlabeled.

Because most of the data is unlabeled, according to [64] and [25], semi-supervised algorithms, in general, operate in the following manner: First, a semi-supervised algorithm is trained using the labeled data to obtain a model that is capable of labeling future data. Then, this model is used to predict the labels of the unlabeled input data. Finally, these predicted label data and the initial labeled data are combined into a new dataset that will be used to train again the same semi-supervised algorithm used initially.

In [45], we have some examples of semi-supervised learning algorithms.

2.4 Principal Components Analysis (PCA)

According to [70, p. 99], reducing the dimension of a set of data is about getting rid of "uninformative information" while maintaining the data that is crucial. PCA focuses on the notion of data that linearly depends on each other (that is, linear dependent) as data that does not contain relevant information [70, pp.99–100].

PCA has applications in many types of problems like classification of datasets, extraction of relevant features (attributes) from the data, and dimensionality reduction of a dataset.

To demonstrate how PCA works, we have a simple example in figure 2.1. This example is based on [3, pp. 158–159]. In this figure, we have a plot with two axes: X_1 and X_2 . These represent the two features from dataset X . We can also see that the samples of dataset X take a form of a stretched circle. Furthermore, we can say that it is much more stretched along the 1st principal component direction compared to the 2nd component direction. In mathematical terms, we refer that the variance of 1st principal component direction is greater than the variance of the direction of the other component. In PCA, these directions are named after the main components (Principal Components).

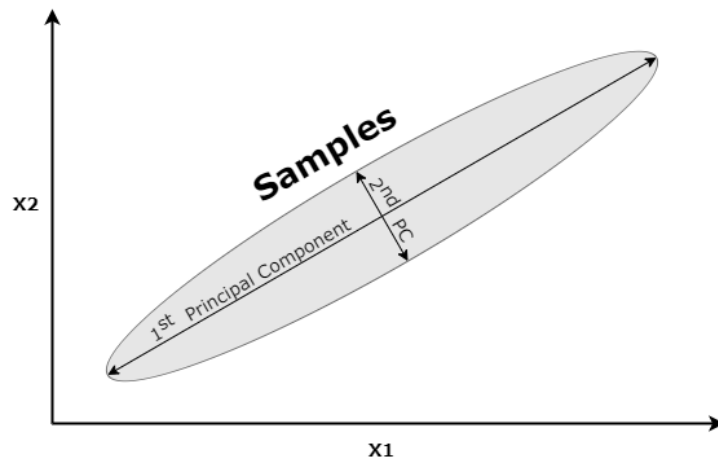


Figure 2.1: PCA with two components (adapted from [3, p. 159])

As stated in [3, p. 159], PCA's strategy, to reduce the number of features of dataset X , is to project all the samples from this dataset, from a 2D plane to a 1D plane (with only one dimension). The data of the second component (PC2) will be lost in the reduction process and, the total number of features of dataset X is reduced to one.

In summary, the basic idea behind the PCA technique is to replace the original redundant features with a reduced number of features that can adequately summarize the information contained in the original dataset [70, p. 101].

Standard PCA is able to reduce linearly separable data (e.g, data that can be separated

using a hyperplane) but with data that is not linearly separable, the linear transformation will not work as well as in the first case. So, to solve this problem, we have to use an extension of the Standard PCA called Kernel PCA (kPCA). This method uses a kernel function to project linearly inseparable data into an higher dimension where it is linearly separable [3, p. 161].

2.4.1 Implementation Examples

In [38], the authors proposed a load estimation system for cloud networks. This system is based on an ESN (echo state network) neural network for the study of the network's temporal evolution. In this case, they used kPCA to reduce the size of the initial database and so, avoid the problem of over-fitting. Overfitting occurs in situations where, the model produced by a neural network or another machine learning algorithm perfectly adapts to the training data and does not consider minor errors in it. We address the overfitting problem in section 5.4, when we implement a multilayer perceptron (see 2.7). According to [38], kPCA was chosen in deterioration of linear PCA, given the former's ability to apply a nonlinear transformation from the input data space to a larger space. With this transformation, kPCA is able to represent and extract the main components of the data set.

Another example is the implementation proposed in article [60]. In this article, the authors proposed PCA as a method for improving the accuracy of an SVM classifier for intrusion detection. In a first stage, PCA had the function of finding the optimal subset of all attributes present in the initial dataset. Then, in a second stage, this subset is used as a training dataset and test dataset for the SVM classifier. After applying this method to a dataset, they achieved better accuracy, detection rate and reduced false alarm rate with the implemented method (PCA + SVM) compared to the stand-alone SVM classifier.

2.5 Random forest (RF)

The Random Forest algorithm uses an aggregation of several different decision trees to make predictions or classify data. The main philosophy behind Random Forest is that the joint use of several decision trees can give more accurate results compared to the use of a single tree.

As we can see in figure 2.2, each decision tree has its own result for the classification of the input data. The result that is obtained by the majority of the trees, will be the final result of Random Forest.

The prerequisites (according to [66]) that have to be fulfilled for Random forest to perform well are:

- the predictions and errors produced by each tree must have a low correlation;
- the features present in the data has to be poorly correlated.

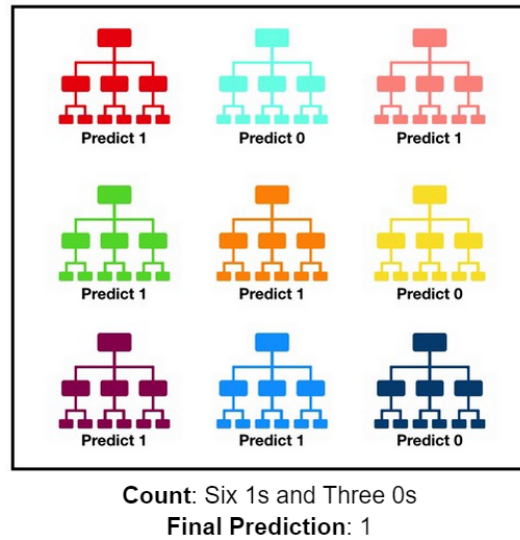


Figure 2.2: Random Forest Decision Scheme (adapted from [66])

To guarantee these prerequisites, strategies such as **Bagging (Bootstrap aggregation)** can be applied. As described in [66], decision trees are very sensitive to changes in the training dataset so this strategy takes advantage of this fact by making each tree choose random samples from the training dataset and use it in their training phase. In the end, Bootstrap aggregation can assure that we have a set of trees that are not correlated with each other (that are different from each other).

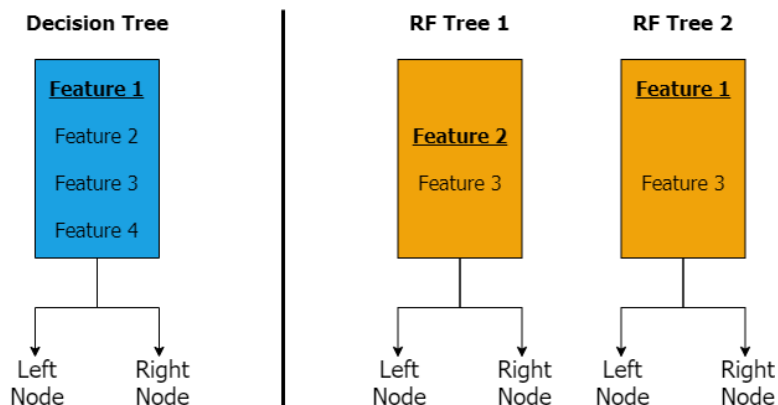


Figure 2.3: Feature Randomness (adapted from [66])

Another strategy used is **Feature Randomness**. In the simple decision tree method (left side of figure 2.3), a node is split using only one feature which is chosen from all the available features in the dataset. As stated in [66], the feature that is considered is the one that produces the most separation between the data in the left node vs. the data in the right node. But in random forest (right side of figure 2.3), we can only consider a

feature that is part of a randomly selected subset of the total number of features in the dataset. This allows for greater variation between the trees and guarantees low correlation between them [66].

2.5.1 Implementation Examples

As an example of a real implementation of this algorithm, we have [33]. The authors proposed a regression model that could be used to estimate the number of containers necessary to satisfy the needs of the network, in a determined time window. In order to choose the machine learning method to create the regression model, a comparison was made between SVM (described in 2.6), Random Forest (described in 2.5) and other methods (presented in [33]) for three load levels of the system: normal load, ladder load (abrupt drop in load for a few minutes and then, an abrupt rise in network load) and load in Zigzag (sudden load spikes). In the end, the authors obtained, on average, better results with Random Forest.

Another example is the article [19]. The authors proposed a model of a IDS (Intrusion Detection System) using a random forest classifier. The main goal was to make an IDS system that could identify and notify the activities of users as normal or abnormal (probably an intruder) based on the network traffic data. The authors implemented Random Forest to classify four different attack types [19].

In the experimental phase, the authors decided to do three implementations and compare them based on certain performance metrics. These three implementations were a single decision tree based classifier, random forest using a feature selection metric named Symmetrical uncertainty (detailed in [19]) and random forest without previous feature selection.

As stated in [19], the performance metrics used to evaluate this implementation were:

- **Accuracy:** Metric described in subsection 2.11.3;
- **Detection Rate:** It is the ratio between the total numbers of attacks detected by the system and the total number of attacks present in the dataset;
- **False Alarm Rate:** It is defined as the ratio between the wrong detection of attack (essentially, a false alarm) and the total number of attacks detected by the system;
- **Mathews Correlation Coefficient:** This metric is defined as the ratio between the observed and predicted binary classifications (more details in [19]).

Results showed that random forest with previous feature selection is on average better than the other two approaches.

2.6 Support Vector Machine (SVM)

SVM is a machine learning algorithm that is commonly used in classification problems but, it can also be used in regression problems. When used in regression problems, SVM is named as SVR (Support Vector Regression).

SVM can deal with two types of situations. The first one is when the data that must be classified is linearly separable (as in figure 2.4). The other one is when the data that must be classified is not linearly separable (as in figure 2.5).

In subfigure 2.4(a), there are several data points associated with two classes: I and NI. There are several possible linear hyperplanes (represented in the figure as dashed lines) that can separate these data points, but linear SVM has to find only one of those. So, the way that linear SVM approaches this problem is by choosing the hyperplane that provides the largest margin between the samples from classes I and NI (subfigure 2.4(b)). This hyperplane serves as the decision boundary between the samples from both classes and, according to [26], it is called maximum marginal hyperplane (MMH). The margin corresponds to the shortest distance from the MMH to the closest data points of both classes. These data points are shown with a thicker border on subfigure 2.4(b), and they are called support vectors.

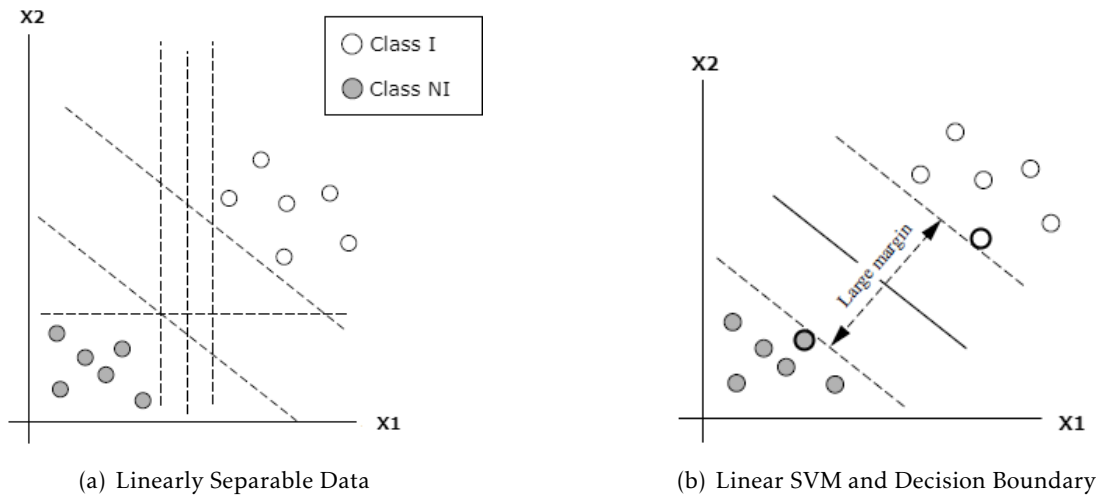


Figure 2.4: Application of Linear SVM on linearly separable data (both figures were adapted from [26, pp. 409–410])

In the case of data that is not linearly separable, linear SVM does not separate very well the two classes, as seen in figure 2.5. So, we have to extend the linear SVM method to a nonlinear one. As stated in [26, p. 413], the idea is that data, which is not linearly separable in a n dimensional space may be linearly separable in a higher-dimensional space. So, what nonlinear SVM does is using a kernel function to obtain the hyperplane that best fits the input data.

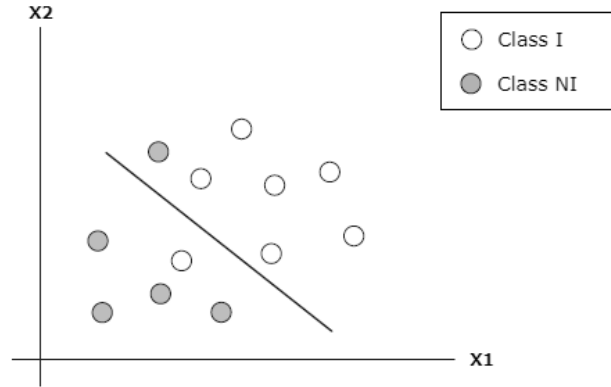


Figure 2.5: Linear SVM with Non-linearly Separable Data (adapted from [26, p. 413])

2.6.1 Implementation Examples

In [35], the authors proposed two different approaches for creating a failure estimation model for a HPC (High Performance Computing) network. The authors tested two approaches: classification and prediction using time-series.

For testing the classification approach, the authors implemented a SVM classifier and compared it with other machine learning methods namely, Random Forest, kNN (K-Nearest Neighbor) and other methods described in [35]. After pre-processing the dataset, the authors tested all the implemented methods based on how well these models could classify these five types of failures: Human error, Software, Hardware, Network and Undetermined. To measure how well these models could classify the dataset, the authors used metrics like Accuracy (described in subsection 2.11.3), RMSE (presented in section 2.8.2) and other metrics described in [35].

In the end, the results showed that SVM obtained good results in terms of accuracy. Hence, this method based model has higher prediction accuracy compared to the other models.

As a second example of implementation, we have [39]. The authors proposed an SVM classifier for fault detection in wireless sensor networks (WSN). SVM is used to define a decision function. This decision function is then executed at cluster heads to detect anomalous sensors. Through some experiments, the authors' goal was to show the effectiveness of the two proposed methods when compared with other classification methods described in [39].

In the experimental phase, the proposed solution was tested with a dataset that contained several common faults that occur in WSNs. Namely, these were gain fault, stuck-at fault, data loss fault, out of bonds fault and random fault. After training all the implementations, the authors used two metrics: Detection accuracy and false positive rate

(described in subsection 2.11.3). These metrics were used to compare all the implementation methods and to see which was the most effective in detecting faults in the dataset. The results showed that the SVM classifier could detect with an accuracy of 90% or above faults that occur with 10% to 50% of probability. The average value of false positive rates of SVM classifier were inferior to the other methods implemented.

2.7 Multilayer Perceptron (MLP)

A multilayer perceptron or feed-forward neural network (FFNN) is an artificial neural network. The reason this network is called feed-forward is that information only flows forward through the network. First, it comes from the input layer, then goes through the hidden layer (s) and finally, arrives at the output layer.

Figure 2.6 shows the structure of the MLP. As we see in the figure, MLP is divided in three main layers: Input layer, Hidden layer and Output layer.

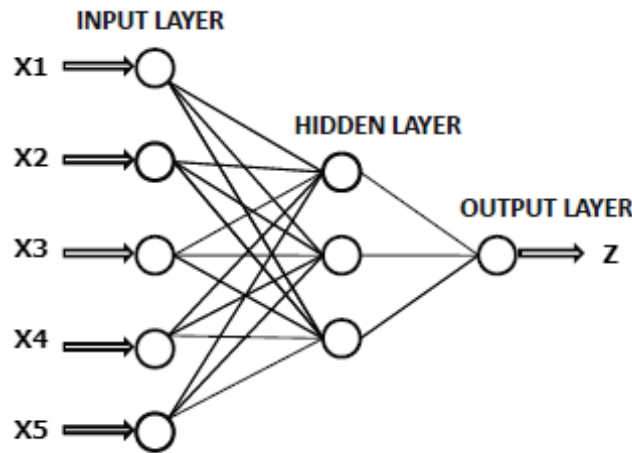


Figure 2.6: Model of an MLP network with one hidden layer (adapted from [1, p 328])

The input layer provides the input data to the MLP network. No computation is performed in this layer. Its main goal is to pass information to the hidden layer(s). The hidden layer is where the multiple computation operations are performed. The MLP can have one or more hidden layers, depending on the problem. The output layer provides the results of all computations done by the network.

Each MLP layer is composed of neurons. These neurons grant a non-linear behavior to the MLP because they have activation functions. These functions can be changed according to the purpose of the network. Depending on the activation function used, the MLP can be used in many different problems, including classification and regression problems (both described in section 2.2). There are several functions that can be used in an MLP (see [40] and [41]) but, the main three used in this dissertation are the Sigmoid, ReLu and Linear functions:

- **Sigmoid:** The sigmoid or logistic sigmoid function is described by equation 2.2, where x represents any variable that can take any real-valued input.

$$\phi(x) = \frac{1}{1 + e^{-x}} \quad (2.2)$$

Figure 2.7 shows the plot of the Sigmoid function. As we can see, the Y-axis is bounded between 0 and 1. Because of this, Sigmoid is often used in problems where the main goal is to predict a probability as an output, since the probability of a certain event is always in the range of 0 and 1.

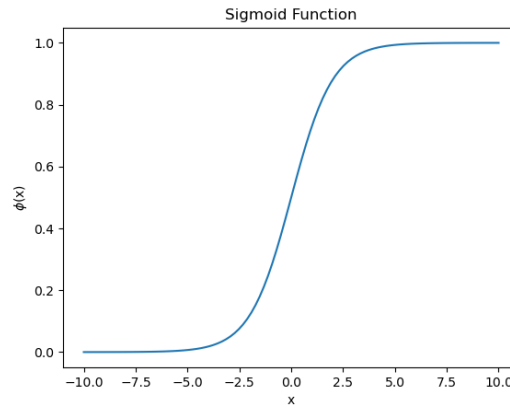


Figure 2.7: Sigmoid Function Plot

- **ReLu:** Rectified Linear Unit or ReLu is a rectified function which means that, unlike Sigmoid, this function is a branch function. ReLu can be described by equation 2.3.

$$\phi(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases} \quad (2.3)$$

As we analyze equation 2.3, we notice that, depending on the value of x , $\phi(x)$ can be either 0 or greater. This means that the output of a ReLu function is always between 0 and $+\infty$. This can be observed in the plot of the ReLu function, which is present in figure 2.8.

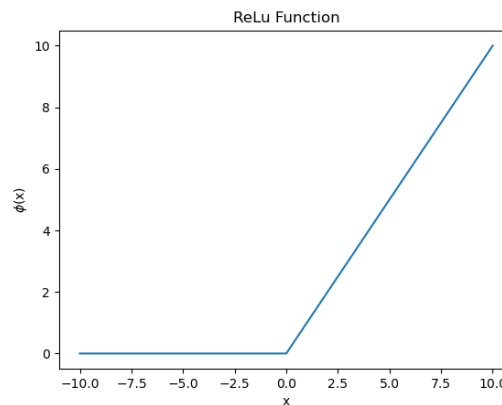


Figure 2.8: ReLu Activation Function

In the context of this thesis, ReLu is used as the activation function in the hidden layers of all MLP implementations presented in sections 5.4, 6.2 and 6.3.

- **Linear:** The linear activation function is the identity function ($\phi(x) = x$), where the input dependent variable x is equal to independent variable $f(x)$. In resume, it means that this function does not changed the input variable value. Figure 2.9 shows the plot of the linear activation function.

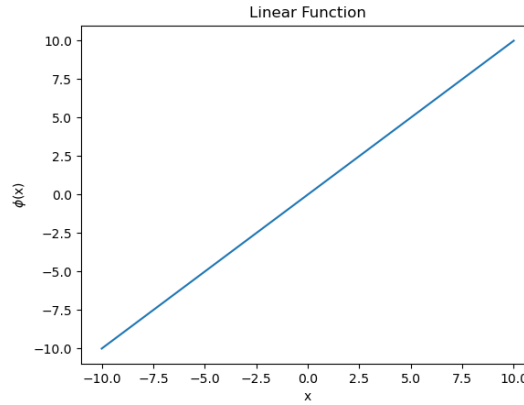


Figure 2.9: Linear Activation Function

As described early in this section, the MLP can be used in many types of problems including classification and regression problems. Until now, we described only a simple MLP configuration with just one hidden layer. When dealing with more complex and harder to solve problems, this configuration might not be enough. There might be the need to add more hidden layers and more units to these layers. This is the main goal behind Deep Neural Networks (DNNs). DNNs are neural networks that use many hidden layers and units to extract more complex features from the input data and improve the performance of these networks. Depending on the input dataset, MLP networks with two hidden layers can even outperform simple MLPs with one hidden layer [61].

2.7.1 Implementation Examples

In [46], the authors implemented and evaluated the performance of several models used for predicting the end-to-end tail latency of microservice workflows on cloud platforms. The methods implemented were Linear regression, SVR, Decision Tree, RF and a deep MLP composed of multiple hidden layers. Initially, the authors selected a few input features for defining the input data of the implemented methods. These features were the average CPU (Central Processor Unit) utilization of load-balanced pods for each microservice, CPU utilization or the CPI (Clock Cycles per Instruction) of VMs that host the pods and number of concurrent clients.

With the final results, the authors concluded that the proposed deep MLP regression model fits the data better than the other models and outperforms all the other models in terms of prediction accuracy.

The second example of an MLP implementation is article [36]. In this case, the authors proposed a model for classifying web services based on a MLP. The methodology was training a MLP with three different algorithms and compare them using the same dataset for training and testing. The three algorithms used were: Backpropagation, Tabu search and Levenberg-Marquardt. All these three learning algorithms are described in detail in [36].

After the training phase, the authors evaluated the three implementations based on metrics like classification accuracy, average recall, average precision and RMSE (described in subsection 2.11.4). Classification accuracy is the same as the accuracy metric presented in subsection 2.11.3. Both the average recall and average precision are just the average values of the recall and precision metrics described in subsection 2.11.3.

The results showed that MLP with Tabu search algorithm outperformed the other two implementations in all the considered metrics.

2.8 Autoencoders (AE)

The basic idea behind the autoencoder is to try to replicate the input data so that the output is as similar as possible to the input.

It seems that the replication of the input is an extremely easy process because the network can just copy the input data and pass it from the input layer to the Hidden Layer(s) and finally, copy it to the Output Layer. However, according to [2, p. 71], this is not possible when we impose a restriction on the number of units that are part of the hidden layer. This imposed "bottleneck" forces the autoencoder to perform a compressed replication of the original input data. As stated in [27], this compression also forces the network to find patterns in the data that can somehow be used to obtain a final result that has a smaller state space. In short, this replication process will inevitably be lossy [2, p. 71]. In the figure 2.10 we can see the structure of a basic autoencoder with the "bottleneck" constraint.

As explained in [2, pp. 71–72], the reduced representation of the data is also called code and the number of units in the hidden layer corresponds to the code size. The phase corresponding to the entry of the data in the input layer plus the processing in the hidden layer, is called encoding (because the size of the data under analysis is reduced, after the encoding phase). In the output phase, the data is reconstructed again to its original dimension. This reconstruction is done based on the reduced version of the input data

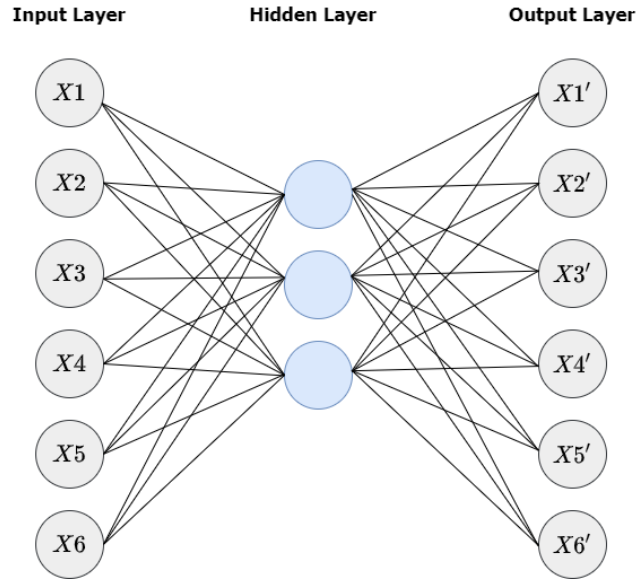


Figure 2.10: Structure of a autoencoder with one hidden layer (adapted from [27])

produced in the encoding phase. This final phase is called decoding.

As problems escalate in complexity, there is a need to increase the number of intermediate layers in the autoencoder. This gave birth to what is called a deep autoencoder (depicted in figure 2.11). According to [2, p. 78] and [23], the increase in the number of intermediate layers can have two main advantages:

- It improves data representation capacity (that is, it improves the autoencoder's ability to retain the same amount of information present at the input but in a more compact dataset);
- It increases the autoencoder's ability to identify data with even more complex patterns, which may be present on the input dataset(s).

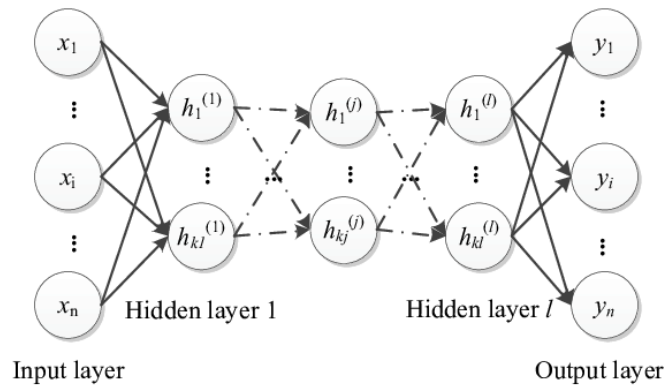


Figure 2.11: Deep Autoencoder with L layers (obtained from [13])

Given their ability to reduce data representation, autoencoders are used for reducing the dimensionality of the data sets under analysis.

2.8.1 Autoencoder Variants

An inherent disadvantage of autoencoders is that "they can simply copy the entire input data to the output, without learning any meaningful representation"[14]. This is because it is not enough to choose the number of intermediate layers and the number of units in those layers. We have to adopt strategies that allow us to restrict the autoencoder's ability to just copy the input to the output. One is the bottleneck strategy that has been described in this section. However, this is not the only one. We also have other variants of autoencoders such as sparse autoencoders (SAE), denoising autoencoders, variational autoencoders, among others (examples and explanation of other methods are presented in [14] and [23]).

These variants have different philosophies from the autoencoders referred in this section. For example, the philosophy behind SAE is that instead of reducing the number of nodes in the hidden layer, restrictions are imposed on the minimum number of units that can be active in the hidden layer(s). This additional restriction will cause the autoencoder to represent the input data through a reduced number of active units (which are units that produce non-zero results on its computations) and, forcing it to discover new patterns within the data. According to [27], an immediate result of this fact is that it allows this neural network to assign each unit present in the hidden layer to attributes present in the data (more details in [27] and [14]).

2.8.2 Implementation Examples

Autoencoders can be used for creating a reduced representation of the original dataset and then, "feed it" to the machine learning algorithm like what was done in [10]. The authors proposed an hybrid approach, which is based on a autoencoder and in a deep neural network, for classification and detection of malicious network traffic, primarily, distributed denial of service(DDOS) attacks.

In the proposed hybrid classification model, the autoencoder is used to learn a compressed representation of the network traffic data. Then, DNN classifies the compressed data using two class labels: Normal or Malicious.

To verify the performance of their approach, the authors considered metrics like F1-score, accuracy, precision and recall (all these metrics are mentioned in subsection 2.11.3). The ROC and Precision-Recall (PR) curves were also used to measure the overall performance of the hybrid classification model. A discussion about ROC curves and how we can acquire information about them is presented in subsection 2.11.3. The PR curve

shows the trade-off between precision and recall for different classification thresholds. A classification threshold represents the boundary between two classes.

The results showed that the proposed approach obtained very high values in all metrics considered. Also, according to [10], the analysis of the ROC and PR curves also suggested that the hybrid classification model is very robust. This is because the false positive rate is very low (close to 0) and the true positive rate is close to 1 (nearly perfect). The PR curve also confirms that the precision and recall scores obtained by the proposed model are very high.

Autoencoders can also be used for anomaly detection or outlier detection based on dimensionality reduction. In [49], the authors describe a nonlinear autoencoder for anomaly detection in a real dataset and in an artificial dataset. As stated by the authors, the logic behind the proposed idea is to train an autoencoder with a previous selected dataset to create a model that fits this same dataset. This model will create a compressed representation of the original dataset. After that, the reconstruction of the original data is done using this compressed representation. The reconstruction error is measured using the RMSE (Root Mean Square). Essentially, RMSE is the square root of the mean of the square of all of the error. The error is the difference between the original data and the reconstructed data. This metric is described, in more detail, in subsection 2.11.4.

According to [49], if the RMSE value begins to increase as the reconstruction process is done, it means that anomalies were found. The authors compared the autoencoder with other methods used for dimensionality reduction like Standard PCA and kPCA (presented in section 2.4) and another variant of autoencoder, that is denoising autoencoder. These comparisons were made by doing some experiments with two datasets: a real dataset and an artificial dataset generated using the Lorenz equations system (further described in [49]). The final results showed that: For the first dataset, Standard PCA had failed to show a significant difference between the anomalies and the normal data but, the other non-linear techniques (kPCA, autoencoder and denoising autoencoder) performed very well by finding most of the anomalies. The authors also noted that denoising autoencoder performs better than the normal autoencoder. For the second dataset, all methods succeeded in detecting anomalies although, denoising autoencoder outperformed the other three techniques.

2.9 Long Short Term Memory (LSTM)

LSTM is a type of RNN (Recurrent Neural Network) that can process entire sequences of data (such as videos or voice). It learns from past experience (long-term). Unlike FFNN units (see section 2.7), RNN units can have feedback loops. This allows the network to take information from prior learning steps in order to use it as input in the next step.

While FFNNs assume that both the inputs and outputs are completely separated and independent from each other, the output of a RNN depends of prior inputs [18].

LSTM was developed to overcome the disadvantage of standard RNNs that "quickly forget what they learn" [12]. Therefore, the purpose of using this neural network is to improve memory capacity by training it to memorize relevant information from past events and "forget" what is not relevant information [12].

In figure 2.12, we can observe a block of memory present in the common LSTM. This is composed of several memory blocks such as the one in the figure.

Each memory block contains memory cells, which make up the LSTM memory part and three regulators: input gate, output gate and forget gate. Depending on the LSTM variant, there are different types of gates and different structures of memory blocks:

- The memory cell (cell in figure 2.12) is responsible for memorizing all dependencies between the input data sequence;
- The input gate (in the figure we have input) controls the input of the block and defines the data that will enter the cell;
- The forget gate defines what data remains in the memory cell and for how long it is stored;
- Finally, the output gate defines which values will be used in the calculation of the estimated value (final result). The result of the calculation will be the estimation of the LSTM based on the input data sequence.

2.9.1 Implementation Examples

This type of neural network, as well as classical RNN, is widely used to estimate the temporal evolution of a system based on time-series data of said system. However, it can also be used in data classification problems.

The first example of application is [5]. In this, the authors intend to estimate the traffic matrix of a network. They further define that "the estimation of a traffic matrix corresponds to the problem of predicting future traffic based on samples from the past". These samples correspond to the collection of traffic data from the network carried out in the past. With the samples of data obtained, the authors decided to choose the implementation of an LSTM instead of the classic RNN as the former is more indicated in the classification, processing and estimation of time series.

In addition to implementing an LSTM, the authors also developed models using classic estimation techniques such as ARIMA and HoltWinters (both described in [5]),

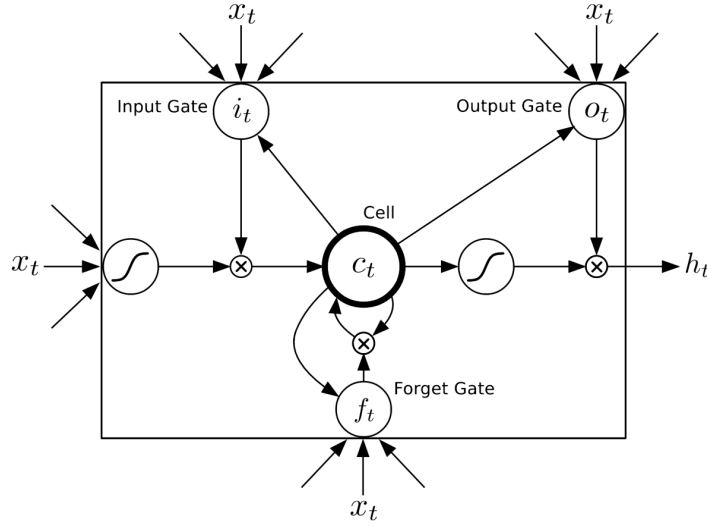


Figure 2.12: Memory Block on LSTM (obtained from [69])

and a FFNN (see section 2.7) for the purpose of comparison and analysis between classical estimation methods and neural networks. In order to assess the accuracy of each model, the authors used as performance metrics the mean square error or MSE. MSE is basically the RMSE metric but without the initial root square so it is just defined as the mean of the squares of the difference between all predicted data and all sample data. MSE and RMSE metrics are described in subsection 2.11.4.

The results obtained favored the LSTM instead of the other techniques, which presented higher MSE values.

The second example of application is described in article [48]. In this, the authors implement an LSTM-autoencoder and an One-class SVM to detect network anomalies in an unbalanced dataset. This hybrid model was proposed with the goal of archiving better results on the detection of anomalies in a SDN (Software-defined Network) by using a lower representation of the input data that is, the result of a LSTM-autoencoder.

The basic idea behind [48] approach is to train an LSTM-autoencoder with normal traffic data (without anomalies) to learn a compressed representation of the input data and learn patterns that can classify this data as normal. Then, the compressed representation is the input data of a One-class SVM. The authors stated that this hybrid model overcomes the shortcomings of the One-class SVM, namely its low capability to operate with large and high-dimensional datasets.

The authors evaluated the proposed model by comparing it to a stand-alone One-class SVM approach (more details in [48]). The final results favored the proposed model in all

the metrics used: F1-score, accuracy, recall and precision. The hybrid model was faster in terms of training time and test time in comparison with the stand-alone One-class SVM approach.

2.10 Hybrid ML models

Hybrid ML models are the combination of two or more different ML models. These models have the potential to produce better results compared to a single ML model. In this dissertation, we combine Linear Regression model and a MLP model to archive better prediction results. This improvement can be seen when comparing with the results archived by the MLP Regressor alone.

Another example of hybrid model is the use of an autoencoder plus another different model like Random Forest. The autoencoder learns how to compress the input data. The resulted compressed data is then, fed to Random Forest to classify this same data into different classes.

2.10.1 Implementation Examples

In [17], the authors propose an hybrid ML model for network intrusion detection. This technique consists of a combination of a multi-layer Autoencoder (deep autoencoder 2.8) and a Deep MLP classifier (MLP with four hidden layers).

As stated in [17], the proposed model is organized in two stages. In the first stage, a deep Autoencoder is used to find the most relevant features in order to maximize the classifier performance. According to [17], "the performance of a classifier highly depends on the selected features". In resume, the main function of the deep Autoencoder in the first stage is to only select the most important features of the input dataset and produce a compressed dataset that only contains these features (The Autoencoder compression process was already discussed in section 2.8). So, the output of the first stage is a compressed version of the input dataset. The second stage is composed of a Deep Neural Network (DNN). This DNN is a four-layer MLP or four-layer feed-forward DNN. The main function of this neural network is to classify the input data in two different classes: "Normal"and "Anomaly".

The authors of [17] evaluated the proposed approach by comparing it to a stand-alone Random Forest classifier and a standalone Deep MLP classifier. The performance metrics used to evaluate the three algorithms were: F1-score, accuracy, recall, precision and False Positive Rate (FPR). All these metrics are described in subsection 2.11.3. The final results favored the proposed hybrid model in all performance metrics used. Finally, the study concluded that the proposed model differs from the other two models because of the use of a deep autoencoder, which performs a latent representation of input data and the

Deep MLP Classifier which has greater generalization capabilities that help improving the performance of the classification decision.

Another example of hybrid models is the joint use of deterministic models and neural networks. In [11], the authors propose an hybrid approach, which combines empirical propagation models and neural networks in order to predict the signal path loss for suburban areas. Empirical models are one type of propagation models that "do not accurately predict the radio waves comportment, depending more on field strength from that specific environment to give an approximation based on measurements." [11]. The empirical propagation models used were Ericsson 9999, Free Space, ECC-33, and TR 36.942 (these models are described in [11]).

According to [11], the problem of predicting the path loss between two terminals can be resumed to finding a function of several inputs and a single output (which is the predicted path loss). The inputs can contain information about the locations of the transmitter and receiver, frequency of the transmitted signals, location of surrounding buildings, etc.

Before implementing the proposed approach, the authors of [11] did a measurement campaign on an area that presented a regular density of vegetation and medium-sized buildings. In [11], this environment is characterized as suburban. The main goal of the measurement campaign was to acquire real measurement data. It was measured the path loss archived by several radio waves, transmitted at 800 MHz and 2600 MHz.

After doing the measurement campaign, the authors calculated the theoretical path loss using all of the four empirical propagation models. According to [11], there were four parameters that were necessary to provide for each model:

- Parameter d , which is the distance between transmitter and receiver;
- Parameter f that represent the frequency (in MHz) of the transmitted signal;
- Parameters h_{TX} and h_{RX} represent the transmission antenna height and reception antenna height, respectively. Both are measured in meters (m);
- Parameter G_r , which is the receiver antenna gain. This parameter was only necessary for EEC model [11].

The proposed hybrid approach is composed of a FFNN neural network, which is trained to learn the difference between the measured path loss and theoretical path loss (called E). The neural network has two inputs: the parameter d and the difference between measured and theoretical path loss. The output is the path loss correction, which

is the difference between the measured path loss (obtained from the measurement campaign) and the difference E . The authors did four different implementations of the hybrid approach. Each uses one of the four propagation models available.

In order to compare the performance of the four hybrid implementations, the authors implemented a simple FFNN neural network, which is trained using the measured data. This network also is trained to predict the path loss archived by the transmitted signals. They also compared the performance of the hybrid approach with another approach that is, the use of only the four empirical propagation models to calculate the path loss. In resume, the authors of [11] compared these three options:

- The hybrid approach, which is composed of four different implementations, each using one of the considered propagation models: Ericsson 9999, Free Space, ECC-33, and TR 36.942 (these models are described in [11]);
- The simple approach, which uses a simple FFNN neural network. This network is trained using the measured data, in order to predict the path loss of a transmitted signal;
- The use of the four empirical propagation models alone in order to predict the path loss of a transmitted signal.

The authors chose the performance metric RMSE (described in subsection 2.11.4) to evaluate and compare the performance of the three options described above.

The final results [11] suggested that the hybrid approach was the better option out of the three because it achieved the lowest RMSE (very close to zero). The results also showed that the simple neural network approach was also a viable option because it obtained good results in terms of RMSE and the predicted path loss is very similar to the measured path loss (for both frequencies). The use of only empirical propagation models to calculate the path loss of transmitted signal showed the worst results in terms of RMSE.

2.11 ML Methods Evaluation

After the implementation of multiple ML methods, we need to evaluate and compare all of them, based on their performances. In order to do this, we used certain metrics, commonly used in classification and regression problems, and specific methods to do our performance evaluation.

We begin this section with the definition of the cross-validation method. Next, we present the notion of the confusion matrix. Finally, we state all evaluation metrics used to evaluate the performance of all methods implemented in chapters 5 and 6.

2.11.1 Cross-Validation

Cross-validation is a statistical method commonly used in machine learning to evaluate ML models. In the context of this thesis, we used K-Fold cross-validation. This method is primarily used in the implementations discussed in sections 5.2 and 5.3.

K-Fold Cross-validation[51] is a method that splits the dataset equally into K groups. One group is selected to test and evaluate the model while the remaining groups are used to train the model. Then, after training and testing the model, a different group is selected to evaluate and test the model. This process is repeated K times until all groups are used for testing the model. After K rounds, the performance of the model is the average of all the performances obtained in all K rounds. When using an Hyperparameter tuner like RandomizedSearchCV[56], each grid combination is used to train a model and then, it is evaluated using a Cross-validation method like, K-Fold. In figure 2.13, we can see the split process done in 5-Fold Cross-validation. In this case, K is equal to 5 so, we have a total of 5 iterations for training the model. Common K values used are 3, 5 and 10.

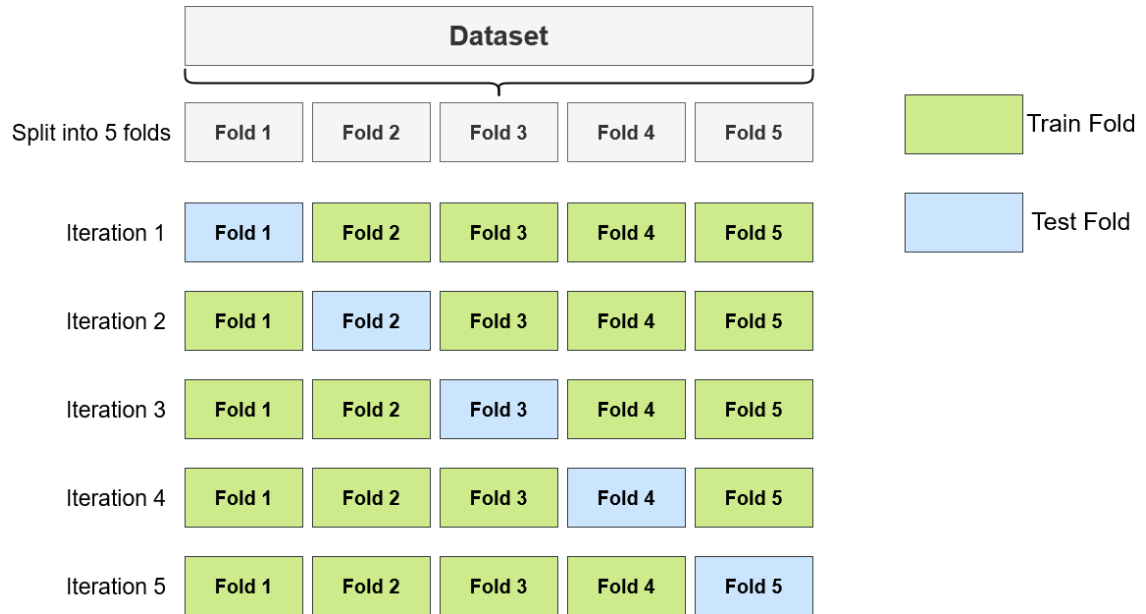


Figure 2.13: Split Process done in 5-Fold Cross Validation

2.11.2 Confusion Matrix

First, in order to distinguish the predictions made by the classification algorithms and to make the explanations of the metrics used to evaluate the classification methods easier, we decided to define the following terms:

- **Positive predictions:** Total number of bookings that were classified as being part of the class NI (Positive Class);

- **Negative predictions:** Total number of bookings that were classified as being part of the class I (Negative Class).

After introducing the above terms, we can now describe the confusion matrix. A confusion matrix is commonly used to evaluate the overall performance of a classifier in a test dataset. In the context of this thesis, we only implemented binary classifiers (classifiers that only consider two classes) so, the structure of the confusion matrix used is like the one present in figure 2.14.

| Confusion Matrix | |
|------------------|--------------|
| True Label | Class I (0) |
| | Class NI (1) |
| Class I (0) | TN |
| Class NI (1) | FN |
| Predicted Label | |

Figure 2.14: General Structure of a Confusion Matrix (adapted from [37])

In order to construct a confusion matrix, it is necessary to know the value of these four classification terms:

- **True Positives (TP):** True positives correspond to the number of positive predictions that the classifier correctly predicted. In this case, these predictions correspond to bookings that are actually part of class NI and were classified as being part of that class;
- **False Positives (FP):** False positives correspond to the number of negative predictions that the classifier incorrectly predicted. In this case, these predictions correspond to bookings that are actually part of class I but, were classified as being part of class NI;
- **True Negatives (TN):** True negatives correspond to the number of negative predictions that the classifier correctly predicted. In this case, these predictions correspond to bookings that are actually part of class I and were classified as being part of that class;

- **False Negatives (FN):** False negatives correspond to the number of negative predictions that the classifier incorrectly predicted. In this case, these predictions correspond to bookings that are actually part of class NI, but were classified as being part of class I.

2.11.3 Evaluation Metrics: Classification

In this subsection, we present the metrics used to evaluate all classification methods implemented in chapter 5.

Accuracy

Accuracy measures the percentage of correctly classified samples in all the predictions made by the classifier. This metric is calculated using equation 2.4, where $n_samples$ corresponds to the total number of predictions made by the classifier.

$$Accuracy = \frac{TruePositives + TrueNegatives}{n_samples} \quad (2.4)$$

Recall

The recall metric tells us how well the classifier is at correctly identifying all class NI samples. In other words, of all class NI samples present in the dataset, how many of these could the classifier correctly identify as being part of this class. Recall is calculated using equation 2.5.

$$Recall = \frac{TruePositives}{TruePositives + FalseNegatives} \quad (2.5)$$

Precision

The precision metric corresponds to the ratio between the true positives and all the positive predictions. In other words, of all the positive predictions made by the classifier, how many are correct. This metric is calculated using equation 2.6.

$$Precision = \frac{TruePositives}{TruePositives + FalsePositives} \quad (2.6)$$

F1-score

F1-score is the harmonic mean between the precision and the recall metrics. In other words, this metric gives the same importance to both precision and recall metrics. It conveys how balanced are the precision and recall obtained by a classifier. This metric is calculated using equation 2.7.

$$F1\text{-score} = 2 \times \frac{Precision \times Recall}{Precision + Recall} \quad (2.7)$$

ROC-AUC

The ROC curve (Receiver Operating Characteristic curve) is a probability curve that shows the overall performance of a classification algorithm at all classification thresholds. A classification threshold represents the boundary between two classes. All values that are greater or equal to the threshold are classified as one class and the other values are classified as another class. The ROC curve is plotted based on two parameters: the true positive rate (TPR) and the false positive rate (FPR). The TPR (or, recall) is presented on the Y-axis of the plot and FPR is on the X-axis. The ROC-AUC is the area under the ROC curve and it measures the classifier's ability to distinguish between two classes (Positive and Negative classes). The higher the AUC value, the better the classification model is at predicting both classes correctly. The AUC value ranges from 0 to 1. The TPR is described by the same equation as the Recall metric (equation 2.5). The FPR metric is described by equation 2.8:

$$FPR = \frac{FalsePositives}{TrueNegatives + FalsePositives} \quad (2.8)$$

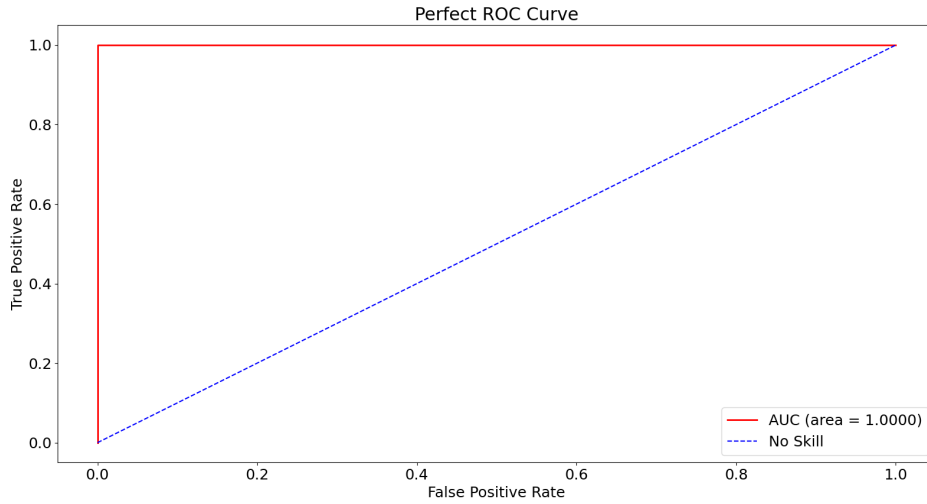


Figure 2.15: ROC Curve

Figure 2.15 shows two curves. The ideal ROC curve achieved by the perfect classifier (red line) and the ROC curve (blue line) achieved by a classifier that has no ability to distinguish between the different classes. We can also see that the ideal classifier has a ROC-AUC value of 1. Also, according to [37], the No Skill curve corresponds to a ROC-AUC value of 0.5.

2.11.4 Evaluation Metrics: Regression

In this subsection, we present the metrics used to evaluate all regression methods implemented in chapter 6.

Mean Squared Error (MSE)

The MSE metric measures the average squared error between the values predicted by a regression model and the actual values. By calculating the square difference, MSE is a metric that penalizes higher differences between actual and predicted values. The lower the MSE value achieved by a regression model is, the better the model is. This is because the difference between actual and predicted values is small. In the opposite side, if a regression model achieves a high MSE, it means that the predicted values are very different from the actual values. In the context of this thesis, we use both MSE and RMSE, in order to penalize models that have higher values in these two metrics. So, we used MSE and RMSE to evaluate the performance of all implementations presented in this chapter. MSE can be calculated using equation 2.9:

$$MSE = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2 \quad (2.9)$$

where: n = Total Number of Data Points in the Dataset
 y = Actual Values
 \hat{y} = Predicted Values

In this dissertation, y represents the true SyncTime of a sample and \hat{y} represents the predicted SyncTime of that sample.

Root Mean Squared Error (RMSE)

The RMSE is the square root of MSE. It is calculated using equation 2.10.

$$RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2} \quad (2.10)$$

Mean Absolute Error (MAE)

The MAE measures the average of the absolute difference between the actual and predicted values. This metric differs from MSE because it calculates the absolute difference and not the squared difference. Another difference is that MAE is a linear metric. This means that all errors are weighted equally on average unlike MSE/RMSE that penalizes large errors (large errors have more weight than small errors). MAE can be calculated using equation 2.11:

$$MAE = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i| \quad (2.11)$$

DATASET ACQUISITION

In this chapter, we give a detailed description of the dataset that was used in this study. In the next section, we describe the data collection process used and scripts developed to automate this process. Finally, we summarize the process of aggregating all acquired data to produce the dataset used.

3.1 Dataset Description

All data was collected from a cluster of three DataMiner agents: Agent 384, Agent 385 and Agent 387. These numbers are the DataMiner agent id or DMAID and they serve to uniquely identify each agent in the cluster. From now on, in order to simplify their identification, we will name them as DMA1, DMA2 and DMA3, respectively.

One of the main functions of these agents is to monitor multiple devices that are part of this cluster and collect information about these devices, which can be used to orchestrate and manage the network. A device or DataMiner device is defined as any device with a network interface. In this dissertation, the software version of DataMiner used was 10.1.2.0-9866 and the SRM framework used was 1.2.10.

The dataset used contains information about a total of 1472 different bookings made between the 7th of May and the 16th of November of 2021. All steps that were taken to acquire this dataset are presented in section 3.2.

The dataset is composed of 19 main columns:

- **Booking ID:** Unique identifier used to identify each booking;

- **Date:** Date when a specific booking request was created. All bookings samples are sorted by Date;
- **Start time:** Time at which the booking should start;
- **SyncTime:** Time difference, in seconds, between the start time and the time in which all the resources requested in a specific booking are finally configured and are available for use. The SyncTime is a discrete parameter and its value must be positive and cannot be zero. It is important to note that this is the parameter that we want to focus our study;
- **Number of Resources:** Number of resources included in a specific booking. A resource exposes a functionality of a device that is available for use. (e.g. A device that can demodulate or modulate a signal can expose two resources/functionality);
- **Number of Concurrent Bookings:** Number of bookings that were scheduled or already "started" at the same time as the booking in analysis;
- **Number of Active Alarms:** This parameter aggregates information about the total number of active alarms in the cluster at a specific date and time;
- **Number of Elements:** Number of elements present in the cluster at a given date and time. An element allows the DataMiner Agents to communicate with devices in order to, for example, acquire specific vendor information of the device;
- **Number of Services:** Total number of DataMiner services that are present at a given date and time in all the agents of the cluster. A service is an aggregation of elements and/or resources and/or other services;
- **Number of Views:** Number of views present in the cluster at a given date and time. It is through the use of views that we can aggregate several elements and/or services in separate tabs. This is done in order to organize hierarchically and distinguish each element in the DataMiner UI (User Interface);
- **Physical Memory Usage [DMAID]:** Memory usage (in percentage) of the agent with id DMAID at a specific date and time. This column type appears three times in the dataset, one for each agent;
- **Total Processor Load [DMAID]:** Total processor load (in percentage) of agent with id DMAID at a specific date and time. In the same manner as the previous column, the dataset has three columns of this type;
- **Ping Src-[DMAID_SRC] Dst-[DMAID_DST]:** Contains the response time in ms (milliseconds) of a communication between two agents. The agent that originated the ping has an id DMAID_SRC and the agent that is the destination of the ping has an id DMAID_DST. There are three possible combinations: Ping Src-DMA1 Dst-DMA3, Ping Src-DMA1 Dst-DMA2 and Ping Src-DMA2 Dst-DMA3.

3.2 Data Collection Process

Before implementing any machine learning algorithm, it is important to collect sufficient data from different sources so that, these algorithms can be trained to produce good results. In this section, it is explained the methodology used to acquire the dataset, described in the previous section.

Before going into the details about the data collection process adopted, it is important to clarify some aspects about the 19 parameters described in the previous section. Some of them were very important to the data collection process while others were merely used for indexing and sorting purposes. So, in order to make the explanations given in this section easier to understand, we divided these parameters in three groups:

- **Index specific parameters:** These parameters were only used for sorting the dataset and to uniquely differentiate each sample. Booking ID, Date and Start time are included in this group;
- **Booking specific parameters:** These parameters describe a specific booking in detail. They are the main source of information that we can acquire about each booking. This group is composed of the Number of Resources and the Number of Concurrent Bookings;
- **Cluster specific parameters:** This group can be further divided in two subgroups. The first subgroup is composed of Number of Active Alarms, Number of Services, Number of Elements and Number of Views. These parameters give us specific information related to the cluster, at a given date and time. The second subgroup is composed of Physical Memory Usage, Total Processor Load and the ping value for each agent. These parameters give us information about the performance of each agent, at a given date and time.

3.2.1 Booking & Cluster State Information

The first step in the data collection process is to know how to acquire information about all bookings that were created in the cluster and how to acquire information about the cluster state, at the time that a booking should start. With this in mind, we developed three main scripts. These allowed us to automate the data collection process.

C# Script 1 - ThesisSilentBooking

One way to create a booking is through the DataMiner main UI. This is normally a manual process that involves selecting a multiple set of necessary parameters, like the number of resources described in the previous section. In order to automate the booking creation process, we developed a C# script with the aid of both the Skyline Communications's System Developer team and Daniela Oliveira (Co-advisor of this thesis). This

script allowed us to automate the booking creation process by setting all the necessary parameters beforehand.

C# Script 2 - CollectRealTimeTrendData

C# script 2 was also developed with the aid of Skyline Communications's System Developer team and the co-advisor of this thesis, Daniela Oliveira. Its main function is to collect the history of all cluster specific and booking specific parameters, in a certain day. The output of this script are multiple files. These can be grouped in the following manner:

- **Booking History:** JSON files that contain information about all bookings created in a specific day. Information about the date, booking start time, booking end time, SyncTime value, booking ID, number of resources included in the booking and many others can be acquired through these files. In resume, each file contains information about multiple bookings created in a specific day.
- **Cluster History:** Excel .csv files that contain information about the number of active alarms in the cluster, number of services, number of elements and number of views for a specific day. Furthermore, it also contains the history of processor load per agent, memory usage per agent and the ping between each agent, in a specific day. In resume, for each parameter mentioned, we have one .csv file. So, for each day, we have 13 .csv files in total.

All files described above are indexed by date and time in order to make the development process of the next script easier. Finally, C# Script 2 is scheduled to run every day at exactly at 19:20 CEST (Central European Summer Time) using the Scheduler feature available in DataMiner.

Python Script

Although, we have the history of each cluster parameter for a certain day, we only need the value of each parameter, at the time the booking is created (Start time). So, in order to filter the parameter history and discover only the value of, for example, the number of active alarms at the time a booking was created, we developed another script. The script was fully developed in Python with the aid of the co-advisor of this thesis, Daniela Oliveira. This script also has the goal of aggregating information. It is used to combine all information that is outputted by C# script 2 in a single .csv file. In section 3.3, we describe the development of this script.

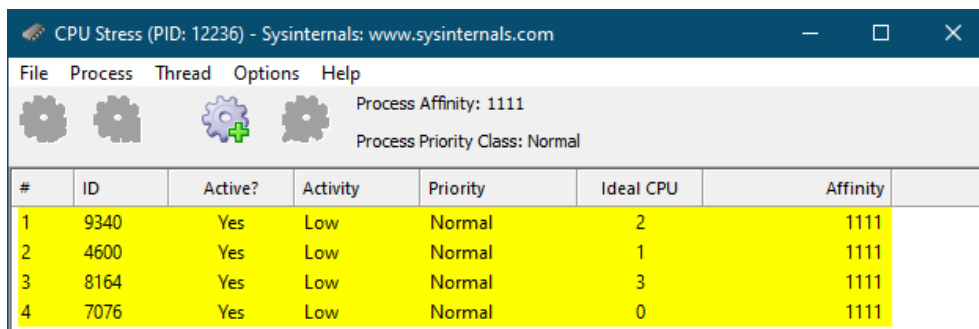
3.2.2 Setups Definition

Now that we know how to acquire the necessary information about the bookings and the cluster state, we need to define several test setups. These setups capture the cluster in different network conditions, such as high processor load in all agents, low processor load in all agents, high memory usage in all agents, etc. So, we defined these to acquire data that describes the cluster in different network conditions.

Depending on the setup, we manually controlled the value of all the booking and cluster specific parameters. The values of the remaining parameters, which were not manually controlled by us, can still change due to certain factors. The first factor is associated with the changes in the total processor load and the physical memory usage of each agent. These parameters can change significantly, depending on the number of events/processes that are running on each DataMiner agent. Another factor is that a parameter's value can be affected due to a change in another parameter. This relation between two parameters can be discovered by analyzing correlation matrices. In section 4.3, we describe them in more detail.

The number of active alarms, number of services, number of elements, and number of views are parameters that can be controlled manually when using DataMiner.

In order to modify the total processor load in each agent, we used a tool called CPUS-TRES [68]. This tool allows the user to simulate high processor load conditions by defining multiple threads running in a loop. The user can also define the load percentage per processor core. Figure 3.1 shows the main interface of CPUSTRES. We can see that four threads were created. Each one is running in a different processor's core. This information can be seen in the Ideal processor column. Each thread process is classified as a low activity process and has normal priority.



| # | ID | Active? | Activity | Priority | Ideal CPU | Affinity |
|---|------|---------|----------|----------|-----------|----------|
| 1 | 9340 | Yes | Low | Normal | 2 | 1111 |
| 2 | 4600 | Yes | Low | Normal | 1 | 1111 |
| 3 | 8164 | Yes | Low | Normal | 3 | 1111 |
| 4 | 7076 | Yes | Low | Normal | 0 | 1111 |

Figure 3.1: CPUSTRES Main Interface

The Physical memory usage per agent was modified by a command-line tool called Testlimit. This tool allows the end-user to allocate a fixed amount of memory (in MB). Testlimit is commonly used "to stress-test your PC and/or applications by simulating

low resource conditions for memory, handles, processes, threads, and other system objects"[47]. Both CPUSTRES and Testlimit were chosen based on [62].

Finally, to control the ping between each agent, we used a tool called Clumsy [57]. Clumsy is a network tool used to degrade the network conditions on Windows systems. It gives the end-user several options (or, functions) to degrade the network like simulating lag in the delivery of a packet, simulate out of order packets, simulating packet drop conditions, etc. The option used was *lag* because it allows us to introduce a certain amount of delay on the delivery of all packets between two agents.

In figure 3.2, we can observe the main interface of Clumsy. In the application's main interface, we can see that there is a field called filtering. This field is used to filter the packets that we want to apply a function, which is, in our case, the *lag* option. In this field, we first specify the IP address of the agent(s) that we want to delay the delivery of packets. Then, we define the delay value in ms.

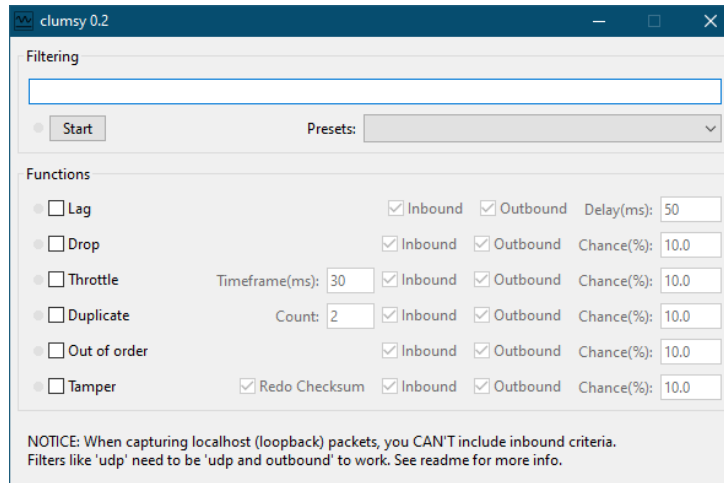


Figure 3.2: Clumsy Main Interface

3.2.3 Dataset Setups

The Dataset used is composed of data that was collected from 13 different cluster setups.

Tables 3.1 and 3.2 show the values of all booking and cluster specific parameters for all dataset setups. Both tables are composed of 17 columns:

- **Setup:** Refers to the setup number;
- **Nº Test:** Refers to the test number of a specific setup;

- **Nº Resources, Nº Concurrent Bookings, Nº Active Alarms, Nº Elements, Nº Services and Nº Views:** These columns refer to some of the parameters already introduced in section 3.1, which are the Number of Resources, Number of Concurrent Bookings, Number of Active Alarms, Number of Elements, Number of Services and Number of Views, respectively;
- **ProcessorLoad_1, ProcessorLoad_2, ProcessorLoad_3:** Refers to the total processor load present in agent DMA1, DMA2 and DMA3, respectively;
- **MemoryUsage_1, MemoryUsage_2, MemoryUsage_3:** Refers to the physical memory usage present in agents DMA1, DMA2 and DMA3, respectively;
- **Ping_All:** Refers to the interval composed of the minimum and maximum ping values observed in all links connecting the three agents. This interval aggregates the values of Ping Src-DMA1 Dst-DMA3, Ping Src-DMA1 Dst-DMA2 and Ping Src-DMA2 Dst-DMA3 (described in section 3.1).

Each setup can be divided into a series of tests. In each test, we changed the value of one or more parameters, depending on the setup. The difference between all setups is the parameters that were manually controlled in each test. In the table below, we can see that, depending on the setup, some parameters have values highlighted in bold. These were the parameters that were manually controlled, to make that specific setup.

All 13 setups present in tables 3.1 and 3.2 can be divided in three groups, depending on their goal. The first group of setups was done with one strategy in mind, which is to control and modify the value of one or more parameters, in order to see their individual effect on the SyncTime value. This group is composed of all setups between 1 and 6. The second group of setups was done to see the effect of changing multiple parameters at the same time in the SyncTime value. Instead of seeing the individual effect of each parameter, we wanted to know the effect of changing multiple parameters per setup. This group is composed of all setups between 7 and 12. Lastly, the third group is composed of only one setup that is setup 13. Setup 13 was done in order to balance the dataset that resulted from the previous 12 setups. We will explain the need to balance the dataset in section 4.1.

3.3 Data Aggregation

After acquiring the data from all setups, it was necessary to aggregate it into a single dataset. This process was fully done by the python script introduced in subsection 3.2.1. As previously explained in that subsection, the output of C# script 2 is a group of several files that contain information that needs to be filtered. This is because we only want to know the values of the cluster and booking specific parameters at the start time of a

Table 3.1: Dataset Setups Table - Part 1

| Setup | N°Test | N°Resources | N°ConcurrentBookings | N°ActiveAlarms | N°Elements | N°Services | N°Views | ProcessorLoad_1 | ProcessorLoad_2 | ProcessorLoad_3 | MemoryUsage_1 | MemoryUsage_2 | MemoryUsage_3 | Ping All |
|-------|--------|-------------|----------------------|----------------|------------|------------|---------|-----------------|-----------------|-----------------|---------------|---------------|---------------|----------------|
| 1 | 1 | 3 | 0 | 5 | 145 | 0 | 15 | 1-8 % | 2-5 % | 5-14 % | 65-66 % | 62 % | 59 % | 0 ms (approx.) |
| | 2 | 3 | 0 | 5-18 | 155 | 0 | 15 | 1-9 % | 1-8 % | 5-16 % | 66 % | 62 % | 59 % | 0 ms (approx.) |
| | 3 | 3 | 0 | 4 | 165 | 0 | 15 | 1-10 % | 1-5 % | 5-18 % | 66 % | 62 % | 59-60 % | 0 ms (approx.) |
| | 4 | 3 | 0 | 4-5 | 175 | 0 | 15 | 1-8 % | 1-4 % | 6-17 % | 66 % | 62 % | 60 % | 0-1 ms |
| | 5 | 3 | 0 | 5-18 | 185 | 0 | 15 | 1-10 % | 2-4 % | 6-15 % | 66 % | 62-63 % | 60 % | 0-2 ms |
| | 6 | 3 | 0 | 4 | 195 | 0 | 15 | 1-13 % | 2-8 % | 4-16 % | 66-67 % | 62-63 % | 60 % | 0 ms (approx.) |
| 2 | 7 | 3 | 0 | 4 | 205 | 0-1 | 15 | 1-8 % | 1-5 % | 5-18 % | 64-66 % | 63 % | 60 % | 0 ms (approx.) |
| | 1 | 5 | 0 | 4-5 | 135 | 0 | 15 | 1-14 % | 1-5 % | 4-15 % | 63 % | 60 % | 58-59 % | 0-1 ms |
| | 2 | 7 | 0 | 8-10 | 135 | 0 | 15 | 1-17 % | 1-4 % | 4-18 % | 63-64 % | 60 % | 58-59 % | 0 ms (approx.) |
| | 3 | 9 | 0 | 5-18 | 135-145 | 0 | 15 | 1-17 % | 1-8 % | 4-16 % | 64 % | 60-65 % | 59 % | 0-1 ms |
| | 4 | 11 | 0 | 4 | 135 | 0 | 15 | 1-11 % | 1-5 % | 5-17 % | 64-66 % | 61-62 % | 59-60 % | 0-1 ms |
| | 5 | 13 | 0 | 4-5 | 135 | 0 | 15 | 1-16 % | 1-6 % | 5-19 % | 65-66 % | 62 % | 59 % | 0-4 ms |
| 3 | 1 | 3 | 0 | 4-5 | 135 | 0 | 15 | 1-9 % | 1-5 % | 5-15 % | 65-66 % | 65 % | 65 % | 0 ms (approx.) |
| | 2 | 3 | 0 | 8 | 135 | 0 | 15 | 1-7 % | 1-4 % | 5-15 % | 70-71 % | 70 % | 70 % | 0 ms (approx.) |
| | 3 | 3 | 0 | 8-10 | 135 | 0-1 | 15 | 1-7 % | 1-4 % | 4-16 % | 80 % | 80 % | 80 % | 0 ms (approx.) |
| | 4 | 3 | 0 | 8 | 135 | 0 | 15 | 1-8 % | 1-6 % | 5-18 % | 90-91 % | 90 % | 90 % | 0 ms (approx.) |
| | 1 | 3 | 0 | 9 | 132-133 | 0 | 15 | 1-4 % | 1-3 % | 3-11 % | 54 % | 65-67 % | 64 % | 12-35 ms |
| | 2 | 3 | 0 | 9 | 133 | 0 | 15 | 1-3 % | 1-2 % | 3-7 % | 54 % | 67 % | 64 % | 21-41 ms |
| 4 | 3 | 3 | 0 | 9 | 133 | 0 | 15 | 2-5 % | 1-3 % | 3-8 % | 54-55 % | 67-68 % | 64 % | 32-49 ms |
| | 4 | 3 | 0 | 9 | 133 | 0 | 15 | 1-4 % | 1-4 % | 4-32 % | 54-57 % | 67-70 % | 64-66 % | 43-62 ms |
| | 5 | 3 | 0 | 9 | 133 | 0 | 15 | 2-4 % | 1-3 % | 3-8 % | 57-58 % | 67-69 % | 64-66 % | 53-75 ms |
| | 1 | 3 | 1 | 9 | 133 | 0 | 15 | 2-7 % | 1-2 % | 4-12 % | 54-55 % | 65 % | 64 % | 0 ms (approx.) |
| | 2 | 3 | 3 | 9-10 | 133 | 0 | 15 | 2-8 % | 1-4 % | 4-15 % | 54-55 % | 65 % | 64 % | 0 ms (approx.) |
| | 3 | 3 | 5 | 9 | 133 | 0 | 15 | 3-13 % | 1-9 % | 5-15 % | 54-57 % | 65-66 % | 63-68 % | 0 ms (approx.) |
| 5 | 1 | 3 | 0 | 9 | 133 | 0 | 15 | 30-34 % | 30-34 % | 32-44 % | 54 % | 68-69 % | 64 % | 0-2 ms |
| | 2 | 3 | 0 | 9 | 133 | 0 | 15 | 39-44 % | 40-44 % | 38-44 % | 54 % | 68-69 % | 64 % | 0-2 ms |
| | 3 | 3 | 0 | 9-10 | 133 | 0 | 15 | 50-56 % | 49-56 % | 50-60 % | 54 % | 68-69 % | 64 % | 0-3 ms |
| | 4 | 3 | 0 | 12 | 133 | 0-1 | 15 | 58-67 % | 61-65 % | 61-72 % | 54 % | 69 % | 64 % | 0-4 ms |
| | 5 | 3 | 0 | 12-13 | 133 | 0 | 15 | 68-98 % | 68-83 % | 68-99 % | 54-57 % | 68-71 % | 64-68 % | 0-4 ms |
| | 6 | 3 | 0 | 9 | 133 | 0 | 15 | 68-98 % | 68-83 % | 68-99 % | 54-57 % | 68-71 % | 64-68 % | 0-4 ms |

Table 3.2: Dataset Setups Table - Part 2

| Setup | N°Test | N°Resources | N°ConcurrentBookings | N°ActiveAlarms | N°Elements | N°Services | N°Views | ProcessorLoad.1 | ProcessorLoad.2 | ProcessorLoad.3 | MemoryUsage.1 | MemoryUsage.2 | MemoryUsage.3 | Ping.All |
|-------|--------|-------------|----------------------|----------------|------------|------------|---------|-----------------|-----------------|-----------------|---------------|---------------|---------------|-------------|
| 7 | 1 | 3 | 0 | 22 - 32 | 133 | 0 | 15 | 29 - 56 % | 32 - 35 % | 33 - 42 % | 66 - 70 % | 64 - 67 % | 62 - 64 % | 13 - 33 ms |
| | 2 | 3 | 0 | 25 - 33 | 133 | 0 | 15 | 44 - 73 % | 40 - 44 % | 39 - 47 % | 68 - 69 % | 64 % | 62 % | 22 - 46 ms |
| | 3 | 3 | 0 | 26 - 30 | 133 | 0 - 1 | 15 | 50 - 62 % | 50 - 54 % | 48 - 77 % | 67 - 70 % | 64 % | 62 % | 30 - 64 ms |
| | 4 | 3 | 0 | 25 - 26 | 133 | 0 - 1 | 15 | 67 - 79 % | 61 - 68 % | 62 - 71 % | 68 % | 64 - 66 % | 62 % | 45 - 84 ms |
| | 5 | 3 | 0 | 26 - 27 | 133 | 0 - 2 | 15 | 80 - 83 % | 73 - 77 % | 73 - 78 % | 68 % | 64 - 65 % | 62 % | 58 - 114 ms |
| 8 | 1 | 3 | 0 | 23 | 133 | 0 | 15 | 35 - 39 % | 31 - 35 % | 32 - 41 % | 67 % | 66 % | 66 % | 12 - 35 ms |
| | 2 | 3 | 0 | 26 - 27 | 133 | 0 | 15 | 43 - 47 % | 38 - 42 % | 40 - 86 % | 70 % | 70 % | 70 % | 22 - 53 ms |
| | 3 | 3 | 0 | 29 - 30 | 133 | 0 | 15 | 54 - 64 % | 50 - 73 % | 48 - 53 % | 75 - 76 % | 75 % | 75 % | 33 - 62 ms |
| | 4 | 3 | 0 | 30 - 31 | 133 | 0 - 1 | 15 | 64 - 70 % | 62 - 77 % | 61 - 67 % | 80 % | 80 % | 81 % | 45 - 111 ms |
| | 5 | 3 | 0 | 30 - 31 | 133 | 0 - 2 | 15 | 79 - 89 % | 71 - 74 % | 72 - 80 % | 85 - 86 % | 85 - 86 % | 85 - 86 % | 58 - 139 ms |
| 9 | 1 | 3 | 0 | 25 - 26 | 143 | 0 | 15 | 31 - 37 % | 31 - 36 % | 34 - 41 % | 66 - 67 % | 66 % | 66 % | 12 - 43 ms |
| | 2 | 3 | 0 | 7 | 153 | 0 | 15 | 40 - 44 % | 41 - 47 % | 39 - 46 % | 71 % | 70 - 71 % | 71 % | 21 - 41 ms |
| | 3 | 3 | 0 | 9 - 10 | 163 | 0 | 15 | 50 - 54 % | 51 - 55 % | 47 - 54 % | 75 - 76 % | 76 % | 76 % | 27 - 55 ms |
| | 4 | 3 | 0 | 10 - 12 | 173 | 0 - 1 | 15 | 56 - 99 % | 64 - 68 % | 60 - 69 % | 81 % | 81 % | 81 % | 45 - 79 ms |
| | 5 | 3 | 0 | 10 - 12 | 183 | 0 - 2 | 15 | 64 - 69 % | 79 - 81 % | 72 - 83 % | 86 % | 85 - 86 % | 86 % | 55 - 116 ms |
| 10 | 1 | 5 | 0 | 6 | 143 | 0 | 15 | 29 - 34 % | 31 - 34 % | 32 - 45 % | 66 - 67 % | 67 % | 66 % | 12 - 31 ms |
| | 2 | 7 | 0 | 9 - 10 | 153 | 0 | 15 | 40 - 44 % | 39 - 43 % | 39 - 51 % | 71 % | 71 % | 70 % | 21 - 49 ms |
| | 3 | 9 | 0 | 11 - 90 | 163 | 0 | 15 | 48 - 56 % | 52 - 70 % | 55 - 64 % | 76 % | 76 % | 76 % | 30 - 69 ms |
| | 4 | 11 | 0 | 12 - 14 | 173 | 0 - 1 | 15 | 60 - 66 % | 65 - 74 % | 61 - 74 % | 81 % | 80 - 81 % | 81 % | 28 - 92 ms |
| | 5 | 13 | 0 | 12 - 14 | 183 | 0 - 2 | 15 | 69 - 76 % | 76 - 81 % | 68 - 73 % | 85 % | 85 - 86 % | 85 - 86 % | 54 - 109 ms |
| 11 | 1 | 5 | 0 | 7 - 8 | 143 | 70 | 65 | 34 - 52 % | 31 - 57 % | 27 - 54 % | 65 - 75 % | 65 - 75 % | 66 - 75 % | 11 - 58 ms |
| | 2 | 7 | 0 | 11 - 12 | 153 | 100 | 95 | 39 - 47 % | 43 - 46 % | 40 - 46 % | 71 % | 70 - 71 % | 71 % | 23 - 63 ms |
| | 3 | 9 | 0 | 11 - 14 | 163 | 130 | 125 | 48 - 60 % | 51 - 56 % | 47 - 53 % | 75 % | 75 - 76 % | 75 - 76 % | 30 - 98 ms |
| | 4 | 11 | 0 | 13 | 173 | 160 - 161 | 155 | 60 - 100 % | 65 - 66 % | 62 - 68 % | 80 - 81 % | 81 % | 80 - 81 % | 43 - 102 ms |
| | 5 | 13 | 0 | 13 - 14 | 183 | 190 - 192 | 185 | 70 - 75 % | 77 - 93 % | 74 - 84 % | 85 - 86 % | 85 % | 85 - 86 % | 55 - 126 ms |
| 12 | 1 | 5 | 1 | 8 - 9 | 143 | 70 - 71 | 65 | 33 - 39 % | 34 - 38 % | 24 - 54 % | 66 % | 66 % | 65 - 66 % | 12 - 42 ms |
| | 2 | 7 | 1 | 11 - 12 | 153 | 100 - 103 | 95 | 41 - 48 % | 43 - 48 % | 41 - 64 % | 70 - 72 % | 70 - 71 % | 71 % | 21 - 48 ms |
| 13 | 1 | 3 | 0 | 4 | 132 | 0 - 1 | 15 | 1 - 7 % | 1 - 18 % | 4 - 17 % | 65 % | 62 % | 60 % | 0 - 1 ms |

booking. So, to filter the unneeded information, we developed the python script. The other goal of this script is to aggregate relevant information into one single file. So, in resume, the python script has two functions: filtering of non-relevant information and aggregating relevant information that comes from the filtering process. The output of the python script is a .csv file.

First, the python script reads multiple JSON files containing information about the bookings that were done on a specific day. For each booking, the script only copies the following parameter values to a .csv file: the date when the booking was created, start time of the booking, booking ID, SyncTime value, DataMiner time of the booking, end time of the booking, and the number of resources that are part of the booking. So, each row of the .csv file is a booking. After this step, the script creates a column called the number of concurrent bookings. In order to discover the number of concurrent bookings when the service starts, the script has to know if two bookings are concurrent. So, for example, to know if two bookings are concurrent or not, the script has to compare the start and end time of booking 1 with the start and end time of booking 2.

Figure 3.3 illustrates a common example of booking concurrency. In the figure below, we have three bookings: booking 1, 2, and 3. Each one of them is represented by a different colored rectangle. The borders of the three rectangles represent the start and end time of that booking. In this explanation, we will consider that all three bookings start immediately, making the SyncTime value of the three bookings equal to 0 seconds. In practice, this never occurs, it only serves to simplify the example explanation. The way that the script knows that if two bookings are concurrent or not is by comparing the start and end time of the two bookings. For example, booking 1 has 0 concurrent bookings but booking 2 has 1 concurrent booking. This is because when booking 1 started, there were 0 bookings created. Because booking 2 is created later than booking 1 and booking 1 is already running when booking 2 is created, booking 1 is considered concurrent to booking 2. In resume, the logic applied is **if booking 2 has a start time that is less or equal to the start time of booking 1 and booking 2 has an end time that is greater than the end time of booking 1 then, bookings 1 and 2 are concurrent.**

The same logic is applied to booking 3 but in this case, booking 3 has 2 concurrent bookings: booking 1 and 2.

After discovering the number of concurrent bookings for each entry in the .csv file, the script needs to acquire the cluster specific parameter values, at the start time of a booking. To do this, it has to read multiple files, which are part of the cluster history group described in subsection 3.2.1 at C# script 2 section.

For example, if we want to know the number of active alarms present in the cluster at 15:35, the script has to filter the number of active alarms history file and search only

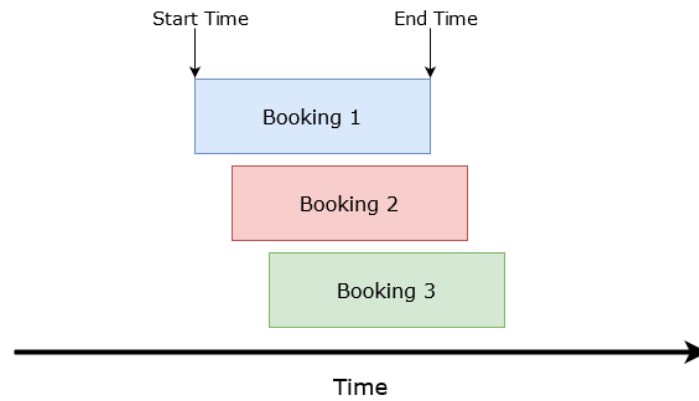


Figure 3.3: Concurrent Bookings Example

for an entry registered at that time. If an entry is found at that time then, the script copies that value to the .csv file. If not, then it must search for an entry that is closer to the time that we want. So, if the file contains an entry registered at 12:00 and then, the next entry is only registered at 17:00, the script copies the value that is in the 12:00 entry. This can be done because the DataMiner software only registers the changes in the parameters values, at the moment they occur. So, if the number of active alarms in the cluster changed at 12:00, an entry is created in the number of active alarms history file. If the file has another entry registered at 17:00, it means the parameter value did not changed until 17:00.

DATA ANALYSIS

In order to gain further insight of the dataset, we decided to use some methods to analyze it. In this chapter, we describe the three methods used: t-SNE (t-distributed Stochastic Neighbor Embedding), CDF (Cumulative Density Function) and Correlation matrix.

4.1 Cumulative Density Function (CDF)

In a first instance, we study the cumulative density function of the SyncTime parameter, in the dataset. The CDF describes the probability that a given variable X has a value less than or equal to a given value x . In the context of this thesis, the CDF describes the probability of a given booking sample to have a SyncTime below or equal to 10 s, for example. The CDF is characterized by the following equation:

$$F_X(x) = P(X \leq x), x \in \mathbb{R} \quad (4.1)$$

In equation 4.1, X represents the SyncTime parameter, x represents an arbitrary value, F_X is the cumulative function of variable X and $P(X \leq x)$ is the probability of variable X be below or equal to an arbitrary value x .

The main goal of the CDF analysis is to study the frequency distribution of SyncTime values in the dataset.

Figure 4.1 shows the plot of the dataset CDF. In the x axis, we have the range of SyncTime values present in the dataset and in the y axis, we have the cumulative frequency of the samples in the dataset. The plot shows that nearly 50% of the total dataset samples have a SyncTime value between 1 and 5 s. This means that the samples of this dataset can be almost equally divided in two groups or classes, depending on the SyncTime value.

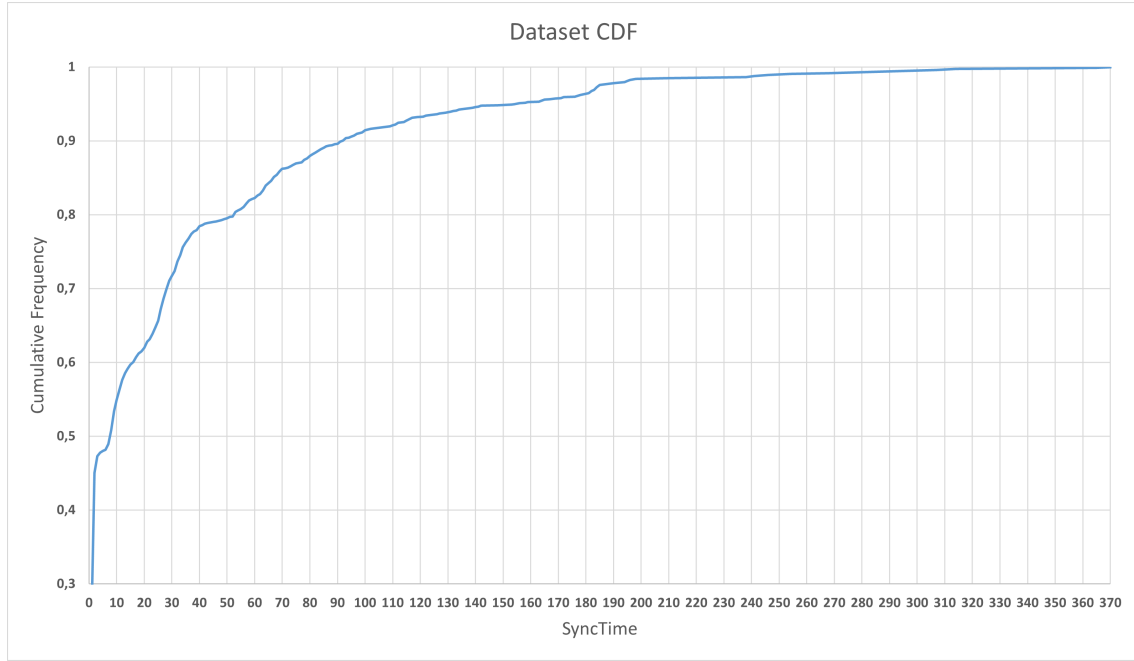


Figure 4.1: Dataset CDF plot

After seeing the dataset CDF, we decided to divide it in two classes, based on the SyncTime value. This process was done in order to begin the implementation of the classification approach. Based on Skyline’s business model and on what the clients of said company consider to be an ideal SyncTime value, we decided to divide the dataset in the following manner: If a booking has a SyncTime below or equal to 5 seconds then, it is classified as I (Ideal). If not, it is classified as NI (Not Ideal).

The following class distribution, presented in figure 4.2, resulted from using 5 seconds as the threshold between the two classes. In this figure, we have a blue bar representing class I and an orange bar representing class NI. The X-axis represents both classes and the Y-axis represents the sample count values. Both numbers on top of both bars represent the total number of samples associated with that bar. From visual inspection of the plot, we can conclude that we have slightly more samples in class NI than in class I. This was expected because in the CDF plot present in figure 4.1, we observed that we could not perfectly divide the dataset using a threshold of 5 s. This is because nearly 50 % of the total samples of the dataset have a SyncTime below or equal to 5 s.

There are several methods that can be used to balance a dataset. Common examples are the oversampling and undersampling techniques. These can be used to balance datasets by creating (oversampling) or removing (undersampling) samples from the original datasets. Reference [22] can be consulted to know more about these techniques. In order to compensate this slightly unbalanced between the samples of both classes, we decided to collect more data, specifically, more data from bookings with a SyncTime below or equal to 5 s. To do this, we used setup 13 which was described in subsection 3.2.3.

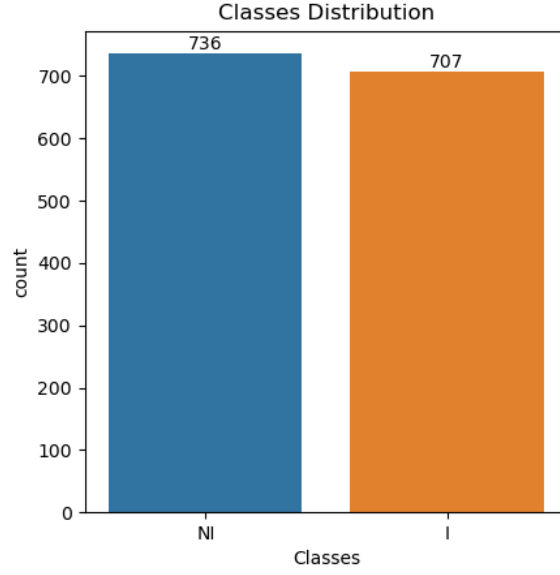


Figure 4.2: Class Distribution in the dataset

Through this setup, we collected information about 29 additional bookings.

4.2 t-SNE: t-distributed Stochastic Neighbor Embedding

To better understand the structure of the dataset, we used a method called t-SNE in order to visualize the dataset on a 2D plot. t-SNE is a method commonly used to visualize high-dimensional data [54].

Figure 4.3 shows the plot of all data points of the dataset used. These points are distinguished by different colors. Each color represents a different setup. We can see that the data points tend to aggregate in little "islands" or clusters. Furthermore, we see that these clusters are also separated from each other. This is caused by the fact that each setup is different in its own accord. Different setups consider different network conditions and also consider different parameter values, as described in section 3.2.3.

As described in section 4.1, we divided the dataset into two classes based on SyncTime value: Class I and class NI. In figure 4.4, we can see clearly the distinction between the two classes. The blue dots represent class I samples and the orange ones represent class NI samples.

In figure 4.4, we can see that almost all orange clusters are located in the top right corner of the figure whereas, the blue clusters are located near the bottom left corner of the figure. From a simple observation of this plot, we can infer that classifiers like SVM, which use hyperplanes to separate the dataset in regions that can be classified into classes, can possibly achieve good results. This is because the clusters of orange dots (class NI) are well separated from the other class I clusters, which facilitates SVM job of finding the

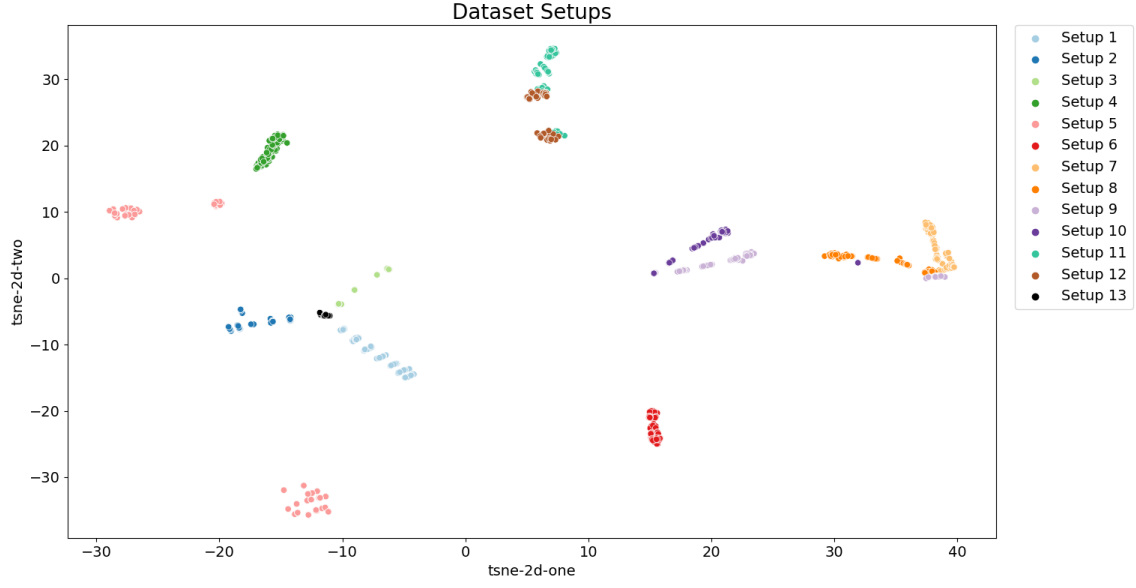


Figure 4.3: 2-Dimensional plot of the dataset

hyperplane that provides the largest possible margin (described in section 2.6).

After coming to these conclusions from a simple observation of the plot, we decided to analyze in detail the clusters where class NI and I data points are packed in a cluster. We highlight two clusters that contained some interesting data points. We named these clusters, A and B, and they are represented in figure 4.4 with black circles.

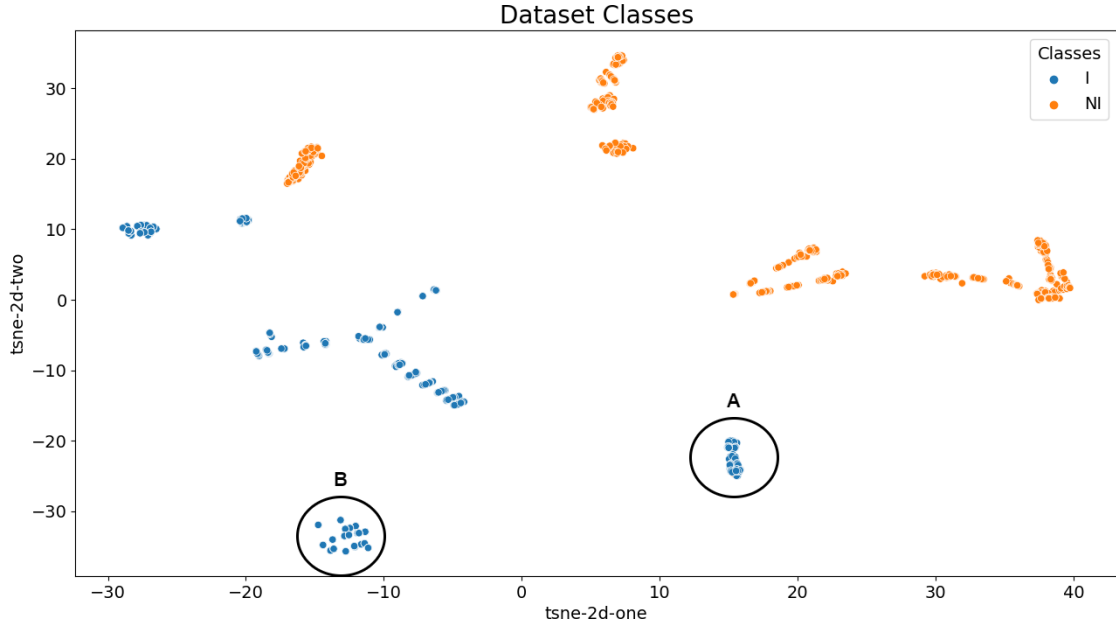


Figure 4.4: Dataset Classes

In order to analyze them, we have figure 4.5 which shows the zoom in view of both clusters. Both clusters contain orange data points that are close to other blue data points.

As we have seen in figure 4.4, the orange points represent data points associated with class NI and blue points represent data points of class I. These data points can be very challenging for classifiers like SVM to correctly classify because, as we have described in section 2.6, SVM uses hyperplanes to separate the data but if this data is not linearly separable (data points of different classes are very close to each other), SVM can have difficulties to find the optimal linear hyperplane that can correctly classify all these data points. In this case, these orange data points present in both clusters can be easily misclassified as false negatives (class NI data points incorrectly classified as a class I data points).

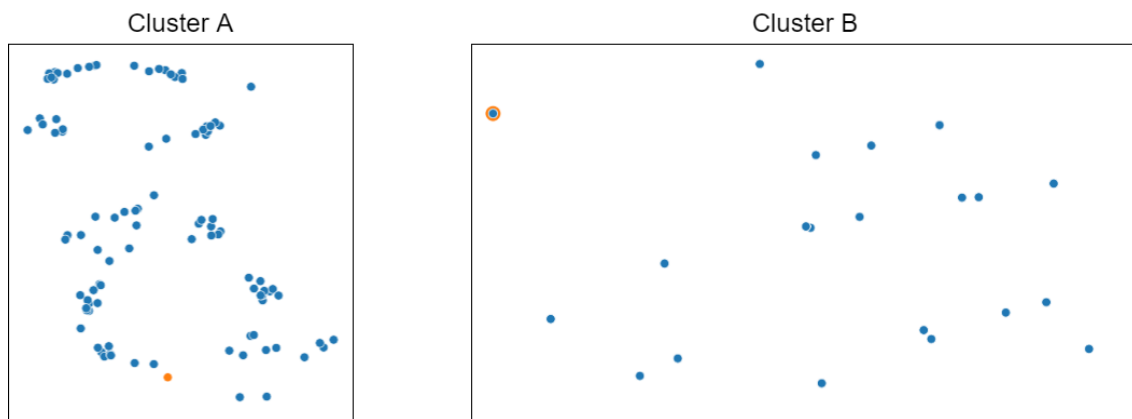


Figure 4.5: Zoom In View of Clusters A and B

4.3 Correlation Matrix

Correlation is a statistical measure of the relationship between two variables. This measure is used to identify if a pair of variables demonstrate a linear relationship between each other. The correlation value can be between -1 and 1. To interpret a correlation matrix, we need to know the meaning of the following three values:

- **-1:** Perfect negative correlation. This correlation value means that the two variables tend to "move" in opposite directions. If one variable increases, the other decreases in the same amount;
- **0:** Null correlation. This correlation value means that the two variables do not have any linear relationship with each other;
- **1:** Perfect positive correlation. This correlation value means that the two variables tend to "move" in the same direction. If one variable increases, the other also increases a proportional amount.

The correlation matrix displays the correlation values of all parameters in a dataset.

Considering the usefulness of the correlation metric in data analysis and to further visualize potential relationships between parameters, we decided to do the correlation matrix for every setup mentioned in subsection 3.2.3. The Excel function, CORREL [34], was used to calculate the correlation values of each matrix. This function allows the selection of two columns to calculate the correlation value between those two. In our case, these columns are each cluster and booking specific parameters (mentioned in section 3.2) as well as, the SyncTime parameter.

Table 4.1 displays the first column of each correlation matrix of the 13 setups. In the context of this dissertation, the first column of a correlation matrix contains the value of the correlation between each of the booking and cluster specific parameters and the SyncTime. This table also shows the color scale that we used to better visualize the different correlation values. In the color scale used, a perfect positive correlation value is represented in green, a null correlation value is displayed in yellow and, a perfect negative value is represented in red.

Before analyzing the table below, we can see that some parameters have a correlation value of #DIV/0!, returned by the CORREL function. According to [34], this error means that the standard deviation of at least one of the parameters of the pair selected to calculate the correlation value is 0. To know what parameters change in each setup, see tables 3.1 and 3.2, which are present on chapter 3.

From the analysis of table 4.1, we had a better understanding of the relation between all eleven cluster and booking specific parameters and the SyncTime. During this analysis, we encountered some interesting correlations. We will analyze these in the following paragraphs.

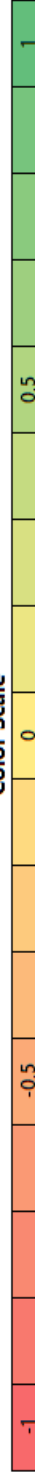
In setup 2 column, the number of resources has a correlation value of 0.81 with the SyncTime. This parameter had the most impact on the SyncTime value of a booking because, it has the highest correlation value comparing to the other 14 parameters. This result indicates that, as the number of resources requested in a booking increases, the SyncTime of said booking increases as well.

Setup 4 is the setup used to study the multiple contribution of the ping between agents in the SyncTime value (review table 3.1 to see the composition of setup 4). As we can see in table 4.1, on setup 4 column, there are three parameters that have a very high positive correlation value: Ping DMA1-DMA3, Ping DMA1-DMA2 and Ping DMA2-DMA3. These high correlations indicate that they have almost a perfect linear relationship with the SyncTime parameter. This, in turn, means that these three can be used to produce a linear regression model that maps the ping between agents to the SyncTime value of a booking. So, we decided to analyze if we can take advantage of this by considering an hybrid approach. This consists in the association of a liner regression model plus an MLP

Table 4.1: Setups Correlation Table

| Parameters | Setups | | | | | | | | | | | | |
|--------------------------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|------|---------|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| N° Resources | #DIV/0! | 0.81 | #DIV/0! | #DIV/0! | #DIV/0! | #DIV/0! | #DIV/0! | #DIV/0! | #DIV/0! | 0.76 | 0.76 | 0.57 | #DIV/0! |
| Ping Src-DMA1 Dest-DMA3 | -0.01 | 0.05 | #DIV/0! | 0.91 | -0.06 | 0.29 | 0.61 | 0.48 | 0.66 | 0.68 | 0.64 | 0.43 | #DIV/0! |
| Ping Src-DMA2 Dest-DMA3 | -0.01 | 0.07 | #DIV/0! | 0.93 | #DIV/0! | 0.16 | 0.61 | 0.50 | 0.69 | 0.71 | 0.61 | 0.31 | -0.04 |
| Ping Src-DMA1 Dest-DMA2 | #DIV/0! | #DIV/0! | #DIV/0! | 0.94 | #DIV/0! | 0.20 | 0.61 | 0.59 | 0.68 | 0.71 | 0.70 | 0.47 | #DIV/0! |
| N° Concurrent Bookings | #DIV/0! | #DIV/0! | #DIV/0! | #DIV/0! | 0.34 | #DIV/0! | #DIV/0! | #DIV/0! | #DIV/0! | #DIV/0! | #DIV/0! | 0.28 | #DIV/0! |
| N° ActiveAlarms | 0.06 | 0.24 | 0.10 | #DIV/0! | 0.04 | 0.67 | -0.17 | 0.57 | -0.33 | 0.20 | 0.61 | 0.59 | #DIV/0! |
| N° Elements | -0.11 | 0.32 | #DIV/0! | 0.42 | #DIV/0! | #DIV/0! | #DIV/0! | #DIV/0! | 0.71 | 0.76 | 0.62 | 0.57 | #DIV/0! |
| N° Services | -0.01 | #DIV/0! | -0.02 | #DIV/0! | #DIV/0! | 0.26 | 0.39 | 0.34 | 0.56 | 0.71 | 0.77 | 0.56 | -0.04 |
| N° Views | #DIV/0! | #DIV/0! | #DIV/0! | #DIV/0! | #DIV/0! | #DIV/0! | #DIV/0! | #DIV/0! | #DIV/0! | #DIV/0! | 0.76 | 0.57 | #DIV/0! |
| TotalProcessorLoad_DMA1 | -0.05 | 0.06 | -0.01 | 0.29 | 0.31 | 0.81 | 0.61 | 0.63 | 0.70 | 0.76 | 0.73 | 0.57 | -0.07 |
| TotalProcessorLoad_DMA2 | 0.02 | 0.14 | 0.10 | -0.05 | 0.30 | 0.75 | 0.64 | 0.59 | 0.71 | 0.75 | 0.77 | 0.54 | -0.01 |
| TotalProcessorLoad_DMA3 | 0.00 | 0.20 | 0.07 | 0.00 | 0.12 | 0.78 | 0.62 | 0.58 | 0.69 | 0.71 | 0.75 | 0.46 | 0.20 |
| PhysicalMemoryUsage_DMA1 | 0.05 | 0.69 | 0.07 | 0.77 | 0.26 | 0.57 | 0.09 | 0.62 | 0.71 | 0.75 | 0.77 | 0.56 | #DIV/0! |
| PhysicalMemoryUsage_DMA2 | -0.07 | 0.77 | 0.07 | 0.43 | 0.25 | 0.44 | 0.00 | 0.62 | 0.71 | 0.76 | 0.76 | 0.57 | #DIV/0! |
| PhysicalMemoryUsage_DMA3 | -0.17 | 0.30 | 0.07 | 0.16 | 0.33 | 0.48 | -0.14 | 0.61 | 0.71 | 0.75 | 0.76 | 0.56 | #DIV/0! |

Color Scale



regression model. The implementation of this approach is called Hybrid Regressor and will be discussed in section 6.3.

Setup 6 column shows that the processor load had a high positive correlation value on a booking's SyncTime value. So, as the processor load in all agents increases, the SyncTime value of bookings increases as well. We also see in this column that, the number of active alarms had also a negative impact on the SyncTime. The number of active alarms and the total processor load are closely related because, as the total processor load in an agent reaches a certain percentage (e.g, 50%), the DataMiner creates an alarm, which gives the user the indication that an agent in the cluster (one or more) is above a certain load percentage. This alarm is then added to the total number of active alarms.

In table 4.1, Setups 9, 10 and 11 columns show the multiple contribution of several parameters on the SyncTime. Almost all parameters had similar correlation values. Reviewing table 3.2, we see that on these setups, several parameters were manually controlled to simultaneously put stress in all cluster agents. There are some parameters that did not have much individual impact on the SyncTime value but, when they were controlled simultaneously with other parameters, their impact became more noticeable. For example, the number of elements did not have the same correlation value in setup 1 column compared to the setups 9, 10 and 11 columns. In setup 1, we manually controlled the number of elements to know the individual contribution of this parameter in the SyncTime of each booking. So, the number of elements did not have a significant impact alone but when we have setups that simulate the cluster in a high load state, the impact of the number of elements in the SyncTime of bookings becomes more noticeable. The same can be said about the number of concurrent bookings. This parameter did not have the same impact in setup 5 compared to setups 9, 10, 11 and 12.

CLASSIFICATION APPROACH: IMPLEMENTATION

In this chapter, we present several ML methods that were implemented with the goal of classifying the booking's SyncTime as I (Ideal) or NI (Not Ideal). These methods are: SVM, Random Forest and MLP.

We begin this chapter by presenting all the data pre-processing steps taken and explain why they were necessary. Following this, we describe each implementation of the ML methods that compose the classification approach. Furthermore, we also describe all discoveries made during these implementations and state all implementation changes that were made because of these discoveries.

5.1 Data Pre-Processing

Before the implementation of any classification algorithm, we had to do several data pre-processing steps. These steps were done in the following order:

1. **Label Encoding:** Label encoding is the transformation of all categorical classes to numeric classes. This step is necessary because none of the methods selected for the classification approach can work with categorical class labels. So, we defined class I as 1 and class NI as 0.
2. **Train-test split:** In order to train both the SVM and RF models, we used `train_test_split` from [55] to split the dataset: 60% of the dataset was used for training the model (called, train set) and 40% for testing these models (called, test set). From this split resulted 4 data subgroups: train data, train class labels, test data and test class labels.

In order to train the MLP, we split the dataset in 6 subgroups: train data, train class labels, test data, test class labels, validation data and validation class labels. The split ratio used was: 60% of the dataset was used for training the model, 20% for validating the model and 20% for testing the model.

3. **Normalization:** Data normalization is essential to the data pre-processing process because, it gives equal weight/importance to each parameter in a dataset. If a set of parameters is not normalized then, a single parameter can increase the classification algorithm performance just because it has bigger values than the other parameters. So, the goal of data normalization is to change the values of numeric features in a dataset to a common scale. In our case, we have some features with different scales: SyncTime, which is measured in seconds and for Total Processor load, which is measured in percentage. To normalize the dataset, we used StandardScaler method[53]. StandardScaler normalizes the values of each feature based on the feature mean (μ) and standard deviation(σ). A value x is normalized to X by using equation 5.1:

$$X = \frac{x - \mu}{\sigma} \quad (5.1)$$

5.2 SVM Implementation

We used the implementation from *scikit-learn* python package [15] to implement SVM.

The parameters (commonly called hyperparameters) that we considered when implementing SVM are:

- **Kernel Type:** Specifies the kernel type to be used by SVM to separate the data. In the context of this dissertation, we test these options: poly (polynomial), radial basis function (RBF), Linear (Poly kernel with degree equal to 1) and Sigmoid;
- **C parameter:** Penalization parameter. Depending on the value of this parameter, we can change the way the incorrect classifications of a SVM classifier are penalized. In other words, as stated in [67], if a low C value is chosen, then incorrect classifications are less penalized, but if a higher value is chosen, incorrect classifications are heavily penalized. Furthermore, when we increase the C value, SVM tries to minimize the penalization of having too many incorrect classifications by choosing a decision boundary with a smaller margin;
- **Gamma:** According to [15], it is a coefficient used in poly, RBF and Sigmoid kernel types. In [30], it is said that the gamma value influences the curvature of a decision boundary. If a high gamma value is used, then the decision curve will be more curved than a low gamma value;

- **Degree:** Specific parameter of the polynomial kernel. It is the degree of the polynomial used as the kernel function.

For tuning all the above hyperparameters, we used RandomizedSearchCV [56] from *scikit-learn* library, and for an implementation reference, we used [21]. First, we define a grid with a group of values for each parameter. This method will test only a limited number of combinations of parameter values in a specific number of iterations. In our case, we considered 460 iterations. We chose this value because, according to [63], we can obtain the top 1% of the best hyperparameters values that can be found in the grid used with 99% confidence. In the final step, RandomizedSearchCV shows the parameters that obtained the best results in terms of F1-score.

It is important to note that RandomizedSearchCV also has the option of using cross-validation (described in section 2.11.1). We decided to use 10-Fold Cross validation.

The grid used to search for the parameters values is presented in table 5.1. We used the same grid in both implementations of SVM. The choice of this grid was based on [21] and [32].

Table 5.1: **Grid used in the SVM implementation**

| SVM Hyperparameters | Tested Values | | | | | |
|---------------------|-------------------|------|---------|-----|------|-------|
| <i>Kernel Type</i> | Polynomial (Poly) | RBF | Sigmoid | | | |
| <i>C parameter</i> | 0.1 | 1 | 10 | 100 | 1000 | |
| <i>Gamma</i> | 0.001 | 0.01 | 0.1 | 1 | auto | scale |
| <i>Degree</i> | 0 - 6 | | | | | |

It is important to refer two things about the grid shown above. The first one is that the values of the C and degree parameters were the same as the implementation done in [21] and, the values of the gamma parameter were the same as the implementation done on [32]. The second thing is that the values for the gamma parameter include two options: auto and scale. In auto option, the value for Gamma is calculated using (5.2). In scale option, it is calculated using (5.3).

$$auto = \frac{1}{n_{feature}} \quad (5.2)$$

$$scale = \frac{1}{n_{feature} * Variance} \quad (5.3)$$

In equation 5.3, *Variance* corresponds to the variance of the input dataset and *n_{feature}* corresponds to the number of features of the dataset.

After doing the tuning of the hyperparameters using the above grid, we obtained the following configurations:

Table 5.2: **Best SVM Hyperparameters**

| Kernel Type | C value | Gamma | Degree |
|-------------|---------|-------|--------|
| Sigmoid | 100 | 0.001 | - |

5.3 RF Implementation

For implementing RF, we used the implementation from *scikit-learn* python package [16].

We decided to consider the following hyperparameters in the tuning process of the RF classifier:

- **n_estimators**: Defines the number of decision trees to be use by Random Forest to classify the samples of the input dataset;
- **max_features**: The number of features to consider when splitting a node. As described in section 2.5, each time the Random forest splits a node, it has to randomly select a subset of features of all the available features in the dataset. The number of features that are randomly selected is defined by this parameter;
- **max_depth**: The maximum depth that any tree can have. According to [20], as the RF trees grow in depth, the more the Random Forest can capture information about the input data and obtain better performance results;
- **min_samples_split**: Represents the minimum number of samples that have to be present at a node so that it can be splitted into more nodes (namely, child nodes);
- **min_samples_leaf**: The minimum number of samples that are allowed at a leaf node. As stated in [16], a node can only be splitted if the child nodes will have at least the minimum number of samples specified in this parameter;
- **Bootstrap**: Refers to the Bootstrap resampling technique used to draw samples and construct all decision trees. If *true* is selected, the samples are drawn using the Bootstrap resampling technique and if not, they are drawn without replacement. For more details about the Bootstrap procedure, see [29] and [7].

The optimization process of the hyperparameters described above was done with the aid of RandomizedSearchCV[56]. The whole optimization process and grid choice were based on [28] and [20]. Furthermore, similar to the SVM implementation (described in section 5.2), we used 10-Fold Cross-validation. Table 5.3 shows the grid used to search for the parameters values, in both implementations.

Table 5.3: Grid used in the RF implementation

| RF Hyperparameters | Tested Values |
|--------------------------|---|
| <i>n_estimators</i> | 1, 2, 4, 8, 16, 32, 64, 100, 200 |
| <i>max_features</i> | 1 - 15 |
| <i>max_depth</i> | 11 evenly spaced numbers between 10 and 110 |
| <i>min_samples_split</i> | 2, 5 and 10 |
| <i>min_samples_leaf</i> | 1, 2 and 4 |
| <i>Bootstrap</i> | True and False |

After doing the tuning of the hyperparameters using the above grid, we obtained the best values for the different RF hyperparameters. These values are presented in table 5.4.

Table 5.4: Best RF Hyperparameters

| Hyperparameter | Value |
|--------------------------|-------|
| <i>n_estimators</i> | 4 |
| <i>max_features</i> | 12 |
| <i>max_depth</i> | 54 |
| <i>min_samples_split</i> | 2 |
| <i>min_samples_leaf</i> | 1 |
| <i>Bootstrap</i> | False |

5.4 MLP Implementation

In the context of this thesis, we tested several MLP configurations using Python and *Keras* ML library was used to implement all MLP networks.

In an implementation of a MLP, there are several degrees of freedom that can be taken into account. These are the following:

- **Activation Function:** Defines the type of activation function to use on the output layer and/or hidden layer(s) of the MLP. In section 2.7, we described the three activation functions used in this dissertation;
- **Loss Function:** This parameter is necessary for the training phase. It computes a value (Loss) that the model should seek to minimize as best as possible during the learning stage. The choice of the loss function is specific to the problem at hand;
- **Number of Hidden Layers:** It defines the number of hidden layers present in the MLP. As seen in section 2.7, this number is not limited to one;
- **Number of Units:** In the case of this dissertation, this parameter defines the number of units per hidden layer.

To implement the MLP, we decided to do the implementation of a simple MLP with 1 hidden layer. We named this implementation MLP classifier. We initially did a total of four tests in order to choose the best unit configuration for the MLP. Table 5.5 resumes the MLP parameters used for each configuration.

The loss plot obtained for each configuration was also analyzed. According to [9], it is through this plot that we can conclude if a model produced by the MLP classifier is overfitting or underfitting the dataset. An overfitted model is a model that achieves an extremely good performance in the training data but, when testing it with new data, it performs extremely poorly. This is because the model learns every detail of the training data that it affects negatively its performance when testing it with new data. Underfitting is the opposite of overfitting. Underfitting means that the model is capable of further learning and possible further improvements during the training phase.

Table 5.5: Configurations Tested for the MLP Classifier

| MLP Parameters | Config. 1 | Config. 2 | Config. 3 |
|------------------------------------|----------------------|-----------|-----------|
| Nº Units Per Layer | 2 | 4 | 6 |
| Activation Function (Hidden Layer) | ReLu | | |
| Activation Function (Output Layer) | Sigmoid | | |
| Loss Function | Binary Cross-entropy | | |

In all three tests done, we decided to use Binary Cross-entropy as the loss function. This is a special case of the cross-entropy loss function, and it computes the cross-entropy loss between predicted classes (which are the output of the MLP classifier) and true classes (which are present in the dataset). It is commonly used in binary classification problems (see subsection 2.2.1 for the definition of binary classification). Binary cross-entropy uses equation 5.4 to compute the loss:

$$Loss = -\frac{1}{N} \sum_{i=1}^N y_i \cdot \log(p(y_i)) + (1 - y_i) \cdot \log(1 - p(y_i)) \quad (5.4)$$

where: N = Total Number of Samples on a Dataset

y_i = Class Label (0 for class I and 1 for class NI) of sample i

$p(y_i)$ = Probability of sample i being part of class I for all N samples

Initially, we implemented configuration 1 MLP and analyzed the loss plot present on figure 5.1. The analysis of it was based on [9]. In the figure we can see two lines. The red line represents the validation loss and the blue line represents the training loss. The X-axis represents the number of epochs of the training phase and, the Y-axis represents the loss/validation loss values.

This plot represents an example of an overfitting situation. This is because the red line (validation loss) continues to increase, while the blue line (training loss) stays almost constant. As we tested the remaining two configurations, this overfitting is increasingly more noticeable (see figures 5.2 and 5.3). So, we decided to discard the other two configurations and focus our attention in configuration 1 MLP.

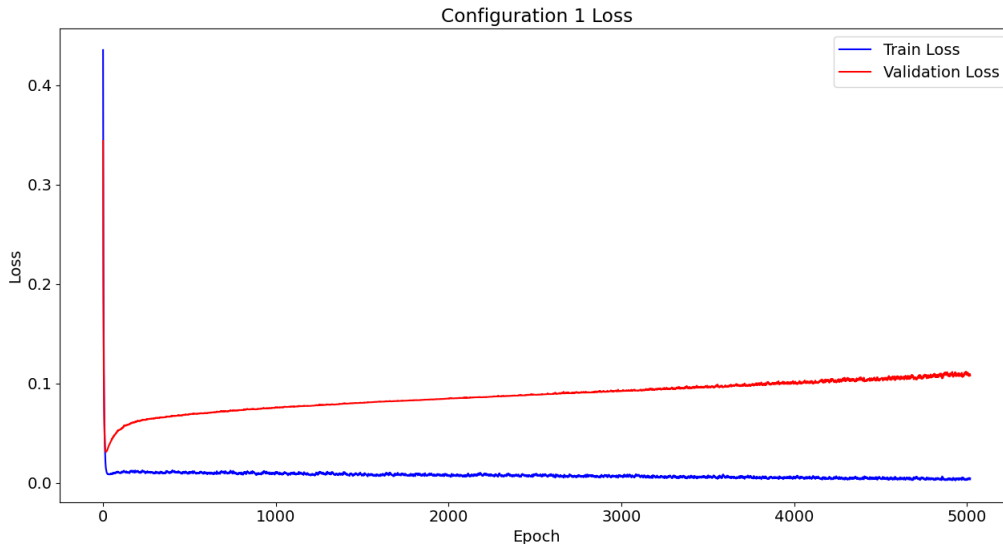


Figure 5.1: Loss plot of Configuration 1

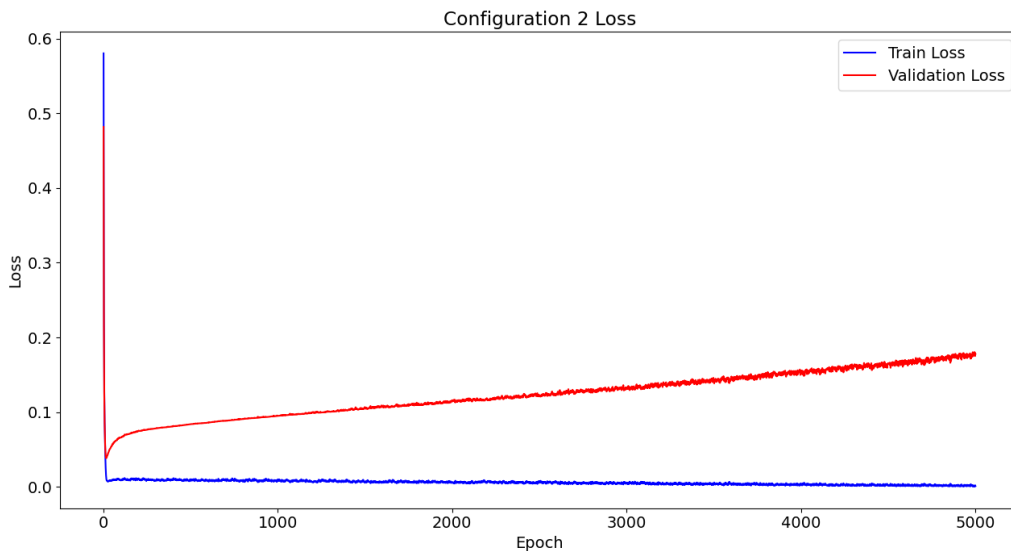


Figure 5.2: Loss plot of Configuration 2

There are many ways to solve overfitting (see more details in [65] and [6]). One possible option is to avoid overfitting by halting the MLP training before the validation loss increases. So, in order to do this, we used the `EarlyStopping` [4] function, which is part of the *Keras* ML library. This function halts the training phase of a neural network, based on the value of the metric that is being monitored. `EarlyStopping` has four important parameters: **monitor**, **mode**, **patience** and **restore_best_weights**.

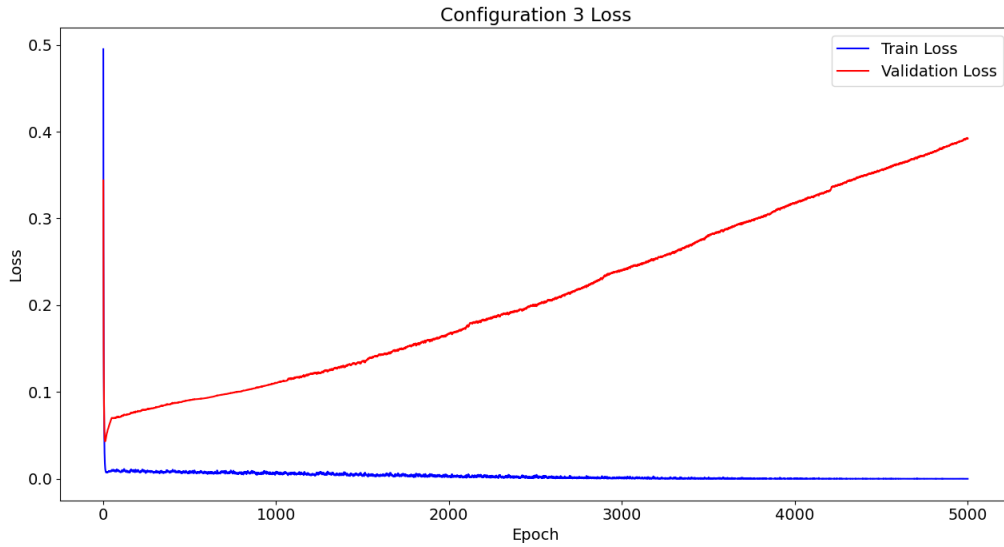


Figure 5.3: Loss plot of Configuration 3

The monitor parameter defines the metric to be monitored. This metric can be metrics like accuracy (described in subsection 2.11.3) or specific quantities like training loss and validation loss. In our case, we chose to monitor the validation loss.

The mode parameter has three possible values: Max, Min and Auto. In Max mode, the training phase is halted when the monitor metric value has stopped increasing. In Min mode, the training phase is halted when the monitor metric stop decreasing. In Auto mode, EarlyStopping infers whichever mode is more suitable to be used, Max or Min, depending on the metric specified in the monitor parameter. In this parameter, we selected Min mode because we want to halt the training of the MLP when the validation loss is not decreasing anymore, and it already achieved the minimum loss value.

The patience parameter allows the user to delay the halt trigger (which stops the training phase of the MLP) by X number of epochs. So, the training phase is halted if after X epochs, the EarlyStopping function sees that there was no improvement in the value of the monitor metric. If X equals to zero and there is no improvement in the value of the monitor metric, the training is halted. The patience value can vary depending on the implementation and dataset.

The restore_best_weights parameter has two options: *True* and *False*. If *True*, the MLP model obtained from the epoch with the best value in the monitored parameter is returned. If it is *False*, the MLP model obtained at the last epoch of training is returned. We use the first option because we want the MLP model obtained in the epoch, where the minimum validation loss was reached.

Figure 5.4 shows the loss and validation loss achieved by configuration 1, when using EarlyStopping. We used a patience value of 10 because it was enough to stop the training

right before the validation loss began to increase. We see that the training is halted at a very early stage, precisely at epoch 18. In turn, this shows that the minimum validation loss was achieved at epoch 18.

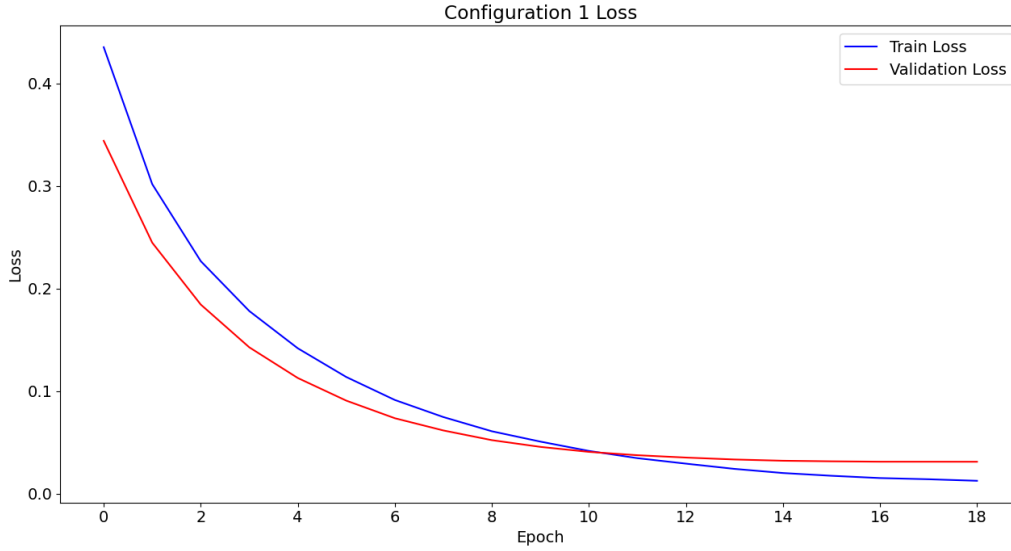


Figure 5.4: Loss plot of Configuration 1 using EarlyStopping

We can see in the figure 5.4 that now the model produced by configuration 1 is slightly underfitting the data. This is because the training loss (blue curve) keeps decreasing. As mentioned in [9], "an underfit model can be identified by the training loss that is decreasing and keeps decreasing at the end of the plot". This is what is observed in the figure. The reason behind the underfit model is because the MLP training was halted too early. But if kept increasing the number of epochs of training, configuration 1 will produce an overfitted model (as seen in figure 5.1). As such, we decided to take another option which is to simply reduce the complexity of the MLP. In other words, we implemented an MLP with only one unit in the hidden layer. Figure shows the loss plot obtained for the MLP implementation with one unit in the hidden layer. Similar to configuration 1, we used EarlyStopping with a patience value of 10 epochs.

When observing the loss plot shown in figure 5.5, it clearly shows that both the train and validation loss curves meet each other during the training process of the MLP. The same did not occur when we described the loss plot of configuration 1 MLP (see figure 5.4). Based on this plot alone, we can see an improvement in terms of model performance, when compared to the previous configurations.

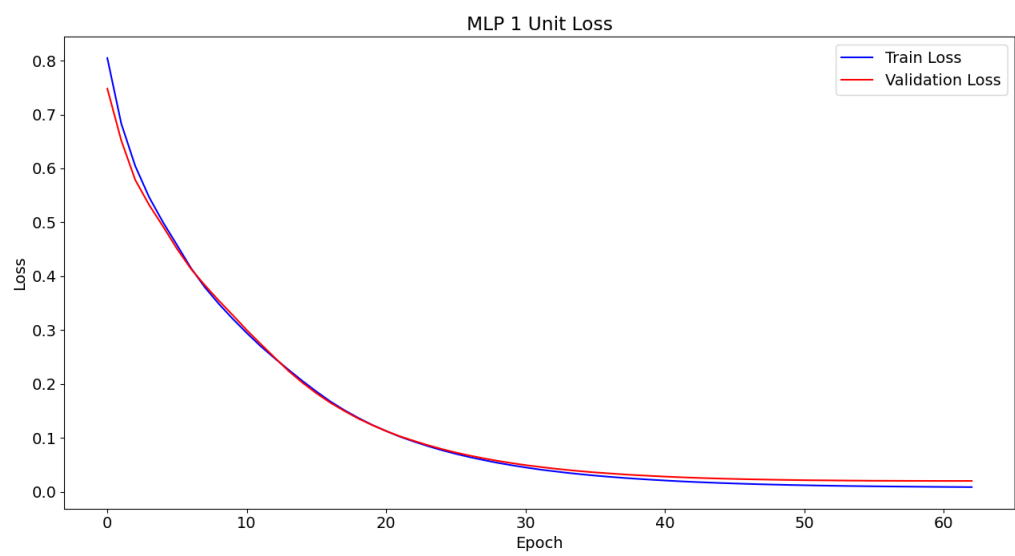


Figure 5.5: Loss plot of MLP with 1 unit

REGRESSION APPROACH: IMPLEMENTATION

In this chapter, we present the implementations of the regression approach. These are: MLP Regressor (see section 6.2) and Hybrid Regressor (see 6.3)

We begin this chapter by presenting all the data pre-processing steps taken before beginning the implementation of any method. Following this, we describe in detail each implementation that compose the regression approach. Furthermore, we also describe all discoveries made during these implementations and state the several optimizations that were made because of these discoveries.

6.1 Data Pre-processing

Similar to what was done in the classification approach, some data pre-processing steps were necessary. These are:

Normalization: Data normalization was done using MinMaxScaler function [52], which is part of *scikit-learn* library. MinMaxScaler rescales each feature data, from the original range to a new given range. The rescaling process is based on the minimum and maximum data values. A feature/variable X is normalized to X_{scaled} by using equation 6.1:

$$X_{scaled} = \frac{X - X_{min}}{X_{max} - X_{min}} \quad (6.1)$$

In equation 6.1, X_{max} and X_{min} are, respectively, the maximum and minimum values of feature X .

Train-test split: In order to implement the MLP Regressor and the Hybrid Regressor, we used `train_test_split` from [55].

The dataset was split in the following manner: 80% was used for training (train set), 10% for validating the model (validation set) and 10% for testing the model (test set).

6.2 MLP Regressor Implementation

The implementation of the MLP Regressor was done based on two different sets of tests. These sets were done in the following order:

- First, an initial set of tests was done with a simple MLP composed of one hidden layer. Details about this implementation are shown in subsection 6.2.1;
- Then, in order to further improve the results achieved by the previous tests, we implemented an MLP with 2 hidden layers. Details about the implementation are described in subsection 6.2.2.

Similarly to the previous MLP implementation, we used EarlyStopping (described in section 5.4) to halt the training of the MLP when the minimum value of validation loss was achieved.

In the following subsections, we present the results obtained in each test and the analysis done of these results.

6.2.1 Initial Test

This initial test is based on four different configuration tests. We did these tests in order to choose the best configuration to implement in the MLP Regressor. We tested the configurations presented in table 6.1. To tune the patience value, we decided to test several values between 100 to 1000. The patience value used was 200 epochs because, by increasing it, we did not see any improvement in any performance metric values achieved by the four tested configurations.

After testing all configurations, we obtained the results shown in table 6.2. We can see that configuration 2 achieved the lowest RMSE and MAE values, followed by configuration 1. Both configurations 3 and 4 showed higher RMSE and MAE values than the other two configurations. In the end, we decided to focus our attention on configuration 2.

Figure 6.1 shows the loss plot obtained for configuration 2 MLP regressor. As we can see, the validation and training loss are steady and do not increase throughout the

Table 6.1: Configurations Tested for the MLP Regressor - Initial Test

| Parameters | Config. 1 | Config. 2 | Config. 3 | Config. 4 |
|------------------------------------|------------|-----------|-----------|-----------|
| N° Units Per Layer | 2 | 4 | 6 | 8 |
| Activation Function (Hidden Layer) | ReLu | | | |
| Activation Function (Output Layer) | Linear | | | |
| Loss Function | MSE | | | |
| Patience | 200 epochs | | | |

Table 6.2: Initial Test MLP Regressor Results

| Performance Metrics | Config. 1 | Config. 2 | Config. 3 | Config. 4 |
|---------------------|-----------|-----------|-----------|-----------|
| MSE | 988.5864 | 971.9015 | 1035.8415 | 1055.6336 |
| RMSE | 31.4418 | 31.1753 | 32.1845 | 32.4905 |
| MAE | 15.0178 | 14.5358 | 16.9965 | 16.7998 |

training. Furthermore, we can also perceive that the gap between the validation curve and the training is small. This an indication that this model is a good fit for the training data. Compared to the perfect fit shown in the loss plot presented in figure 5.5, configuration 2 MLP does not produce a perfect fit.

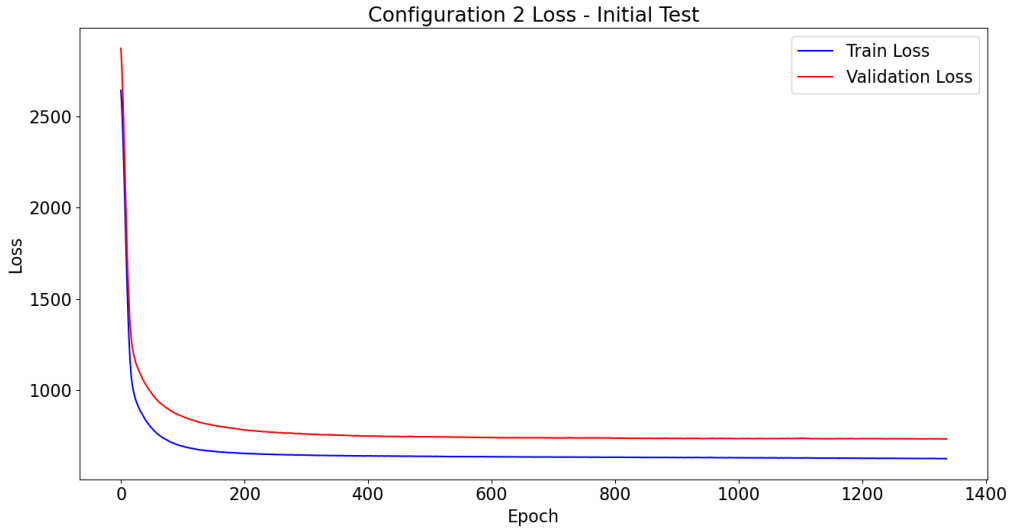


Figure 6.1: Loss plot of Configuration 2 MLP Regressor - Initial Test

In figure 6.2, it is presented the distribution of predicted values and actual values obtained for the configuration 2 MLP Regressor, based on two views. Subfigure 6.2(a) shows a general view of the distribution of predicted and actual values and subfigure 6.2(b) restricts the previous plot to show only the predicted SyncTime values below 10 s. In both subfigures, the y-axis represents the predicted SyncTime and the x-axis represents the actual SyncTime. Both axis are measured in seconds. The blue dots represent a pair

of values: predicted and actual SyncTime. Predicted SyncTime is a decimal value so, in order to compare both the actual and the predicted SyncTime values, we always round the predicted value to the nearest integer number.

Both plots show how well the MLP Regressor model is at predicting the test samples. If the predicted value and the actual value are equal then, the correspondent blue dot of this pair will be located in the black diagonal. In other words, a perfect regression model has all blue dots located in the black diagonal.

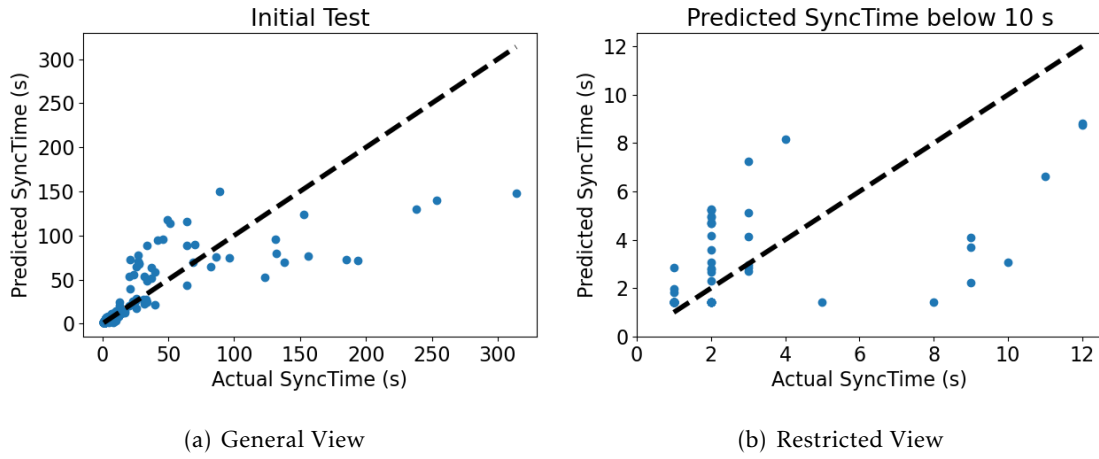


Figure 6.2: Predicted vs. Actual SyncTime values for MLP Regressor - Initial Test

In subfigure 6.2(a), we can see that the model can't accurately predict samples that have a SyncTime above 150 s. This is because all actual values above or equal to 150 s are being predicted as being below 150 s. Furthermore, we can see that there are no predictions above 150 s and the maximum predicted SyncTime is 150 s. In conclusion, the MLP regressor showed difficulties in predicting booking samples that have a SyncTime above 150 s.

Subfigure 6.2(b) shows that the model respected the SyncTime's lower bound of 1 s. This means that the configuration 1 MLP did learn that the SyncTime's lower bound is 1 s and did not make any predictions below this value.

6.2.2 2 Hidden Layers Test

As described in section 2.7, an MLP with more than one hidden layer, depending on the input data, can achieve better results by acquiring more high-level and complex features from the input data. So, to further improve the results obtained in the previous tests, we decided to test the addition of one more hidden layer to the MLP. We implemented four different MLP configurations (see table 6.3) and we chose the best configuration based on the RMSE and MAE values achieved by each configuration.

Table 6.3: Configurations Tested in the 2 Hidden Layers Test

| Parameters | Config. 1 | Config. 2 | Config. 3 | Config. 4 |
|------------------------------------|------------|-----------|-----------|-----------|
| N° Units | 6-6 | 7-7 | 8-8 | 9-9 |
| Activation Function (Hidden Layer) | ReLu | | | |
| Activation Function (Output Layer) | Linear | | | |
| Loss Function | MSE | | | |
| Patience | 700 epochs | | | |

Table 6.4 summarizes the results achieved by each configuration.

Table 6.4: 2 Hidden Layers Test Results

| Performance Metrics | Config. 1 | Config. 2 | Config. 3 | Config. 4 |
|---------------------|-----------|-----------|-----------|-----------|
| MSE | 989.1383 | 1019.3781 | 948.4013 | 964.2601 |
| RMSE | 31.4506 | 31.9277 | 30.7961 | 31.0525 |
| MAE | 15.5475 | 15.7685 | 14.1915 | 14.2095 |

Configuration 3 has the lowest RMSE and MAE of all tested configurations. A lower RMSE means that, on average, the squared differences between the actual and predicted SyncTime values are lower compared to the other configuration models. A lower MAE also shows that the absolute differences are, on average, lower compared to the other models. In resume, both metrics indicate that configuration 3 produced a estimation model that better fits the train data and presented on average, lower errors.

Configuration 1 and 2 achieved the worst results in terms of RMSE and MAE, which means that these configurations obtained models that have large errors between the predicted and actual SyncTime.

As described in subsection 2.11.4, we want to choose regression models that present lower MSE/RMSE values and penalize models that have large errors between the actual and predicted SyncTime. Based on the analysis of the results in table 6.4, we chose configuration 3 to implement in the final implementation of the MLP Regressor. The performance results of this implementation are analyzed in section 7.2.

Figure 6.3 shows the loss plot obtained for configuration 3. As we can see in the plot, the validation and training loss are steady and do not increase throughout the epochs of training, which means this model is a good fit for the train data. Furthermore, we can see that this loss plot is similar to the one presented in figure 6.1.

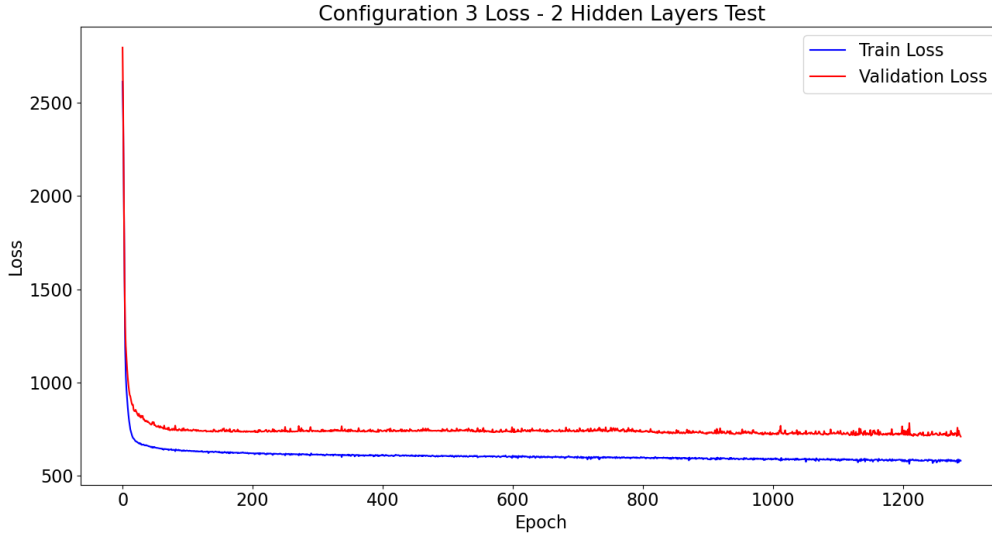


Figure 6.3: Loss plot of Configuration 3 MLP Regressor - 2 Hidden Layers Test

6.3 Hybrid Regressor Implementation

As seen in section 4.3, the average ping between all agents has a very high correlation with the SyncTime parameter. The Hybrid Regressor is an attempt to try to take advantage of this.

This implementation is divided in two parts. In the first part, we implemented a linear regression model that can predict a SyncTime of a booking based on the average ping in all agents. In the second part, we implemented an MLP Regressor in order to predict the difference between the predicted SyncTime from the linear regression model (LR) and the actual SyncTime (A). We called this difference, difference error (E), which is given by equation 6.2:

$$E = A - LR \quad (6.2)$$

where: E = Difference Error

A = Actual SyncTime Value

LR = Linear Regression Model Predicted Value

E is a real variable, unlike the SyncTime value, which is a nonzero positive variable.

Regarding the first part, we implemented a linear regression model using Linear-Regression function, which is available in the *scikit-learn* library [50]. We decided to use the average ping in all agents (it is the average value between the Ping Src-DMA1 Dst-DMA3, Ping Src-DMA1 Dst-DMA2 and Ping Src-DMA2 Dst-DMA3) as input to the linear model.

Figure 6.4 shows the linear regression model (blue line) produced from the average ping in all agents and the SyncTime samples from Setup 4. In this setup, the average ping in all agents has a correlation value of 0.95 with the SyncTime parameter.

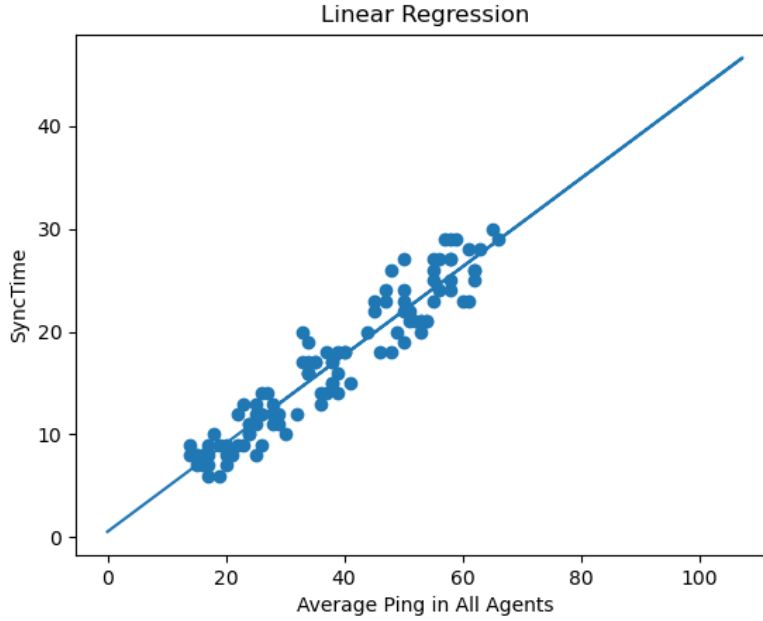


Figure 6.4: Linear Regression Model

The obtained linear regression model is described by equation 6.3:

$$y = 0.4292.x + 0.5485 \quad (6.3)$$

where: y = Predicted SyncTime value
 x = Average Ping in All Agents

In the second part, we implemented an MLP composed of two hidden layers to estimate E . In order to configure the number of units in each hidden layer, we implemented and tested the following four configurations, presented in table 6.5.

Table 6.5: Hybrid Regressor Configurations

| Parameters | Config. 1 | Config. 2 | Config. 3 | Config. 4 |
|------------------------------------|-------------|-----------|-----------|-----------|
| N° Units | 6-6 | 7-7 | 8-8 | 9-9 |
| Activation Function (Hidden Layer) | ReLu | | | |
| Activation Function (Output Layer) | Linear | | | |
| Loss Function | MSE | | | |
| Patience | 1000 epochs | | | |

Table 6.6: Hybrid Regressor Results

| Performance Metrics | Config. 1 | Config. 2 | Config. 3 | Config. 4 |
|---------------------|-----------|-----------|-----------|-----------|
| MSE | 1017.0931 | 1041.6455 | 894.3314 | 1052.7051 |
| RMSE | 31.8919 | 32.2745 | 29.9054 | 32.4454 |
| MAE | 15.5882 | 15.3199 | 13.4897 | 15.8180 |

Table 6.6 shows the results achieved by each configuration for $A = E + LR$.

Configuration 3 achieved the best results in terms of RMSE and MAE. This shows that configuration 3 obtained a estimation model that has all squared and absolute differences between the actual and predicted SyncTime lower compared to the remaining configurations models on table 6.6. This, in turn, means that the configuration 3 obtained a estimation model that better fits the train data.

Configurations 1 and 2 presented interesting results. The RMSE of configuration 2 is higher than configuration 1 RMSE but, in turn, the MAE of configuration 2 is lower than the MAE of configuration 1. This is interesting because, it tells us that configuration 2 model was more penalized in terms of large errors (higher RMSE) compared to configuration 1 model but, on average, the absolute differences/errors of configuration 2 model are lower (lower MAE) than configuration 1 model.

Lastly, configuration 4 achieved the worst results in terms of all performance metrics used.

Based on the analysis of these results, we chose configuration 3 to be the final configuration of the Hybrid Regressor. The results of the implementation of this configuration are analyzed in section 7.2.

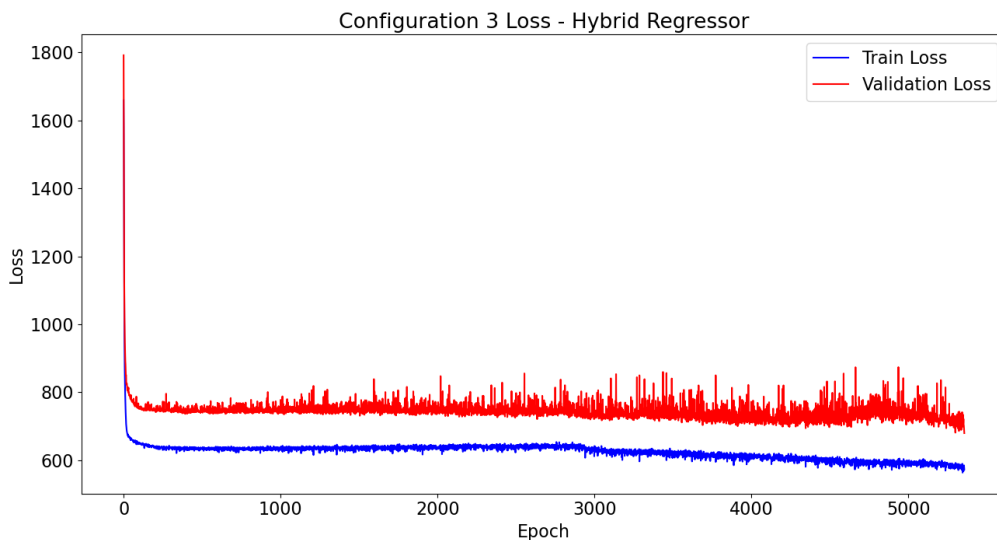


Figure 6.5: Loss plot of Configuration 3 Hybrid Regressor

After obtaining these results, we decided to analyze the loss plot obtained for configuration 3. Figure 6.5 shows the loss plot obtained. As we can see, the validation and training loss are steady until, near epoch 3000, the training loss begins to decrease rapidly while the validation loss decreases more slowly. This is an indication that the model is beginning to overfit. In fact, if we look at figure 6.6, we see that the model is beginning to overfit if we increase the number of epochs of training. We can state this because, as the number of epochs increases, the validation loss increases rapidly. Similar to what was described in section 5.4, EarlyStopping halted the training before the model started to overfit. More specifically, EarlyStopping halted the training at the epoch where the validation loss was minimum.

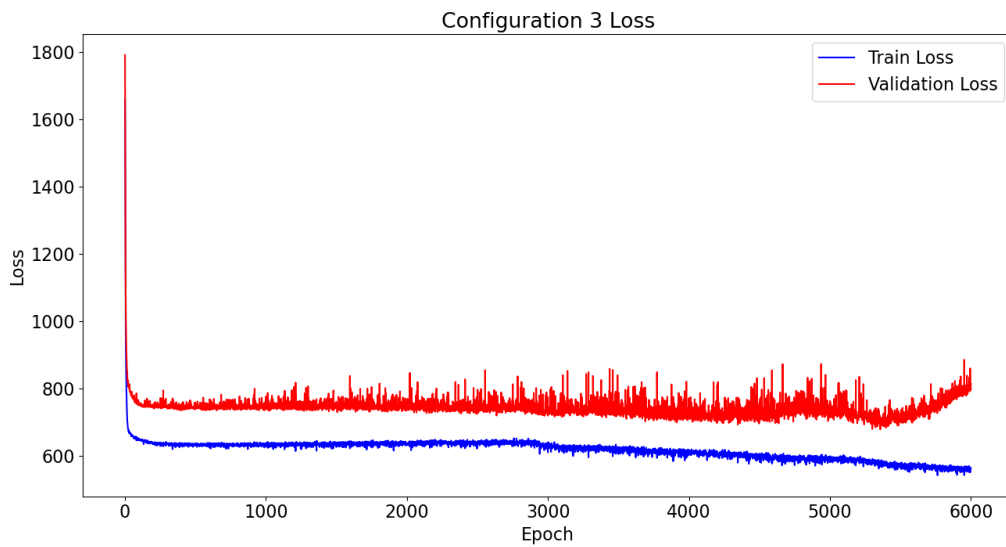


Figure 6.6: Loss plot with Overfitting effect

DISCUSSION OF RESULTS & DEVELOPED SOFTWARE

In this chapter, we present and discuss the best performance results achieved by the implementations described in chapters 5 and 6. Then, we describe the developed software, which is an additional contribution of this thesis.

7.1 Classification Approach

In this section, we present the best performance results of the implementations described in chapter 5.

Table 7.1 shows the best results achieved by all classification implementations.

Table 7.1: Classification Approach Best Results

| Classification Metrics | SVM | RF | MLP Classifier |
|------------------------|--------|--------|----------------|
| Accuracy | 0.9983 | 0.9983 | 1.0000 |
| F1-Score | 0.9983 | 0.9983 | 1.0000 |
| Precision | 1.0000 | 1.0000 | 1.0000 |
| Recall | 0.9966 | 0.9966 | 1.0000 |
| ROC-AUC | 0.9968 | 0.9983 | 1.0000 |

We can see that the MLP obtained perfect results in all performance metrics used. In turn, the SVM and RF obtained slightly lower results compared to the MLP. Both the SVM and RF achieved equal results in terms of F1-score, accuracy, precision and recall. The only difference between them is the ROC-AUC value. RF achieved a higher ROC-AUC value than the SVM, which means that the first classifier is slightly better at distinguishing between the two classes.

It is important to refer that both the SVM and RF did not correctly predict all NI samples. This can be inferred by looking at the confusion matrices obtained by both implementations. Figures 7.1 and 7.2 show the confusion matrices obtained by SVM and RF, respectively. Figure 7.3 shows the confusion matrix obtained by the MLP Classifier.

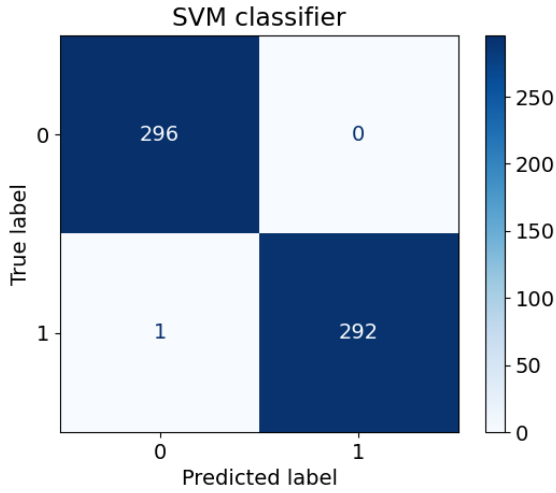


Figure 7.1: SVM Confusion Matrix

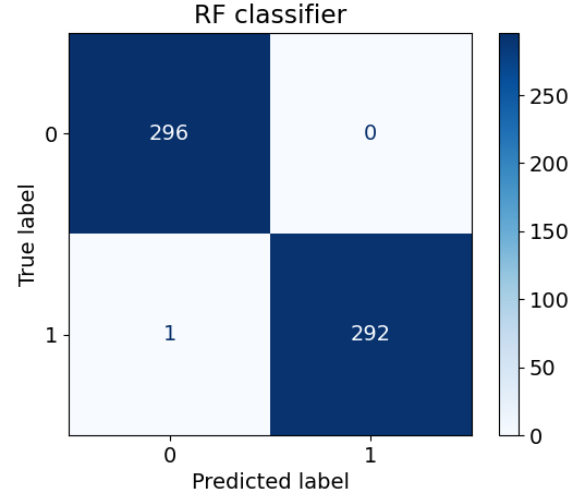


Figure 7.2: RF Confusion Matrix

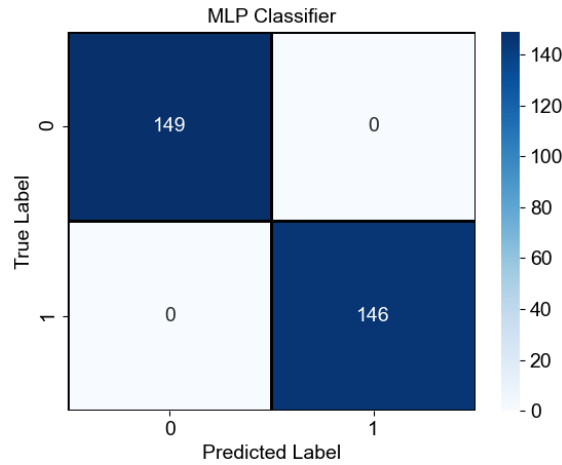


Figure 7.3: MLP Classifier Confusion Matrix

We can see in the SVM and RF confusion matrices that both these implementations obtained one false negative. As described in section 4.2, there are two problematic class NI data points, which are present in two clusters of the dataset: A and B. In both clusters, there were class NI data points that were too close to class I data points. The existence of these two data points can be the cause of the existence of false negatives in both implementations. In turn, the MLP Classifier obtained zero false negatives and false positives which justify the perfect precision and perfect recall values obtained.

7.2 Regression Approach

In this section, we present the best performance results of the implementations described in chapter 6. Table 7.2 resumes the results achieved by the MLP Regressor and the Hybrid Regressor implementations.

Table 7.2: **Regression Approach Best Results**

| Performance Metrics | MLP Regressor | Hybrid Regressor |
|---------------------|---------------|------------------|
| MSE | 948.4013 | 894.3314 |
| RMSE | 30.7961 | 29.9054 |
| MAE | 14.1915 | 13.4897 |

The Hybrid Regressor showed the lowest RMSE and MAE values of the two implementations. A lower RMSE value indicates that the squared differences between the actual and the predicted samples of the MLP Regressor model are slightly lower than the squared differences obtained by the MLP Regressor model. This means that the first model was not as much penalized by large errors as the latter model. In turn, a lower MAE indicates that, on average, the absolute differences between the actual and predicted samples of the Hybrid Regressor model are lower compared to the ones obtained by the MLP regression model. To further see how well both implementations are at predicting samples, we analyze figures 7.4 and 7.5.

Figure 7.4 shows the distributions of predicted and actual SyncTime samples of the MLP Regressor (subfigure 7.4(a)) and Hybrid Regressor (subfigure 7.4(b)). Figure 7.5 shows a restricted view of the previous figure. It shows only the distribution of predicted SyncTime values below 13 s.

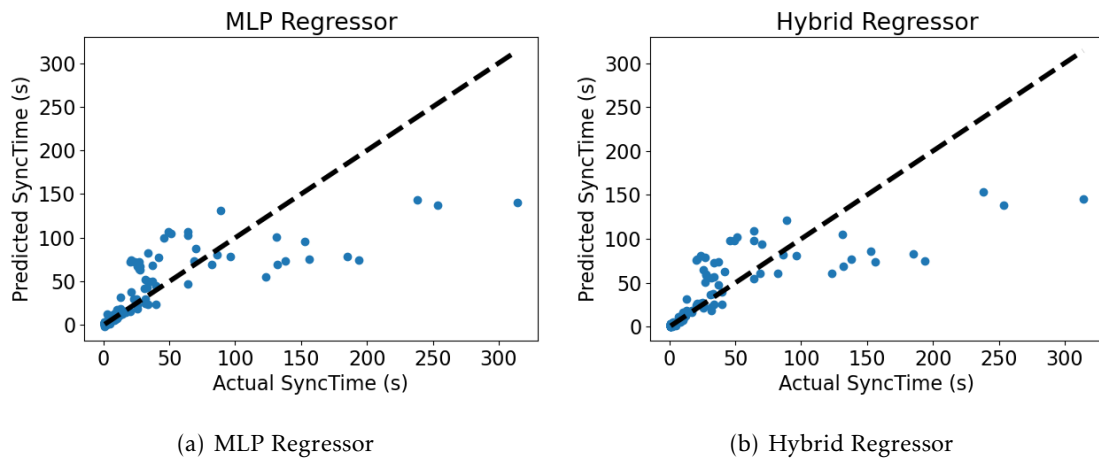


Figure 7.4: Predicted vs. Actual SyncTime values

In subfigures 7.4(a) and 7.4(b), we can see that both the MLP and Hybrid Regressor models still have the limitation present on the initial test model (this model was described

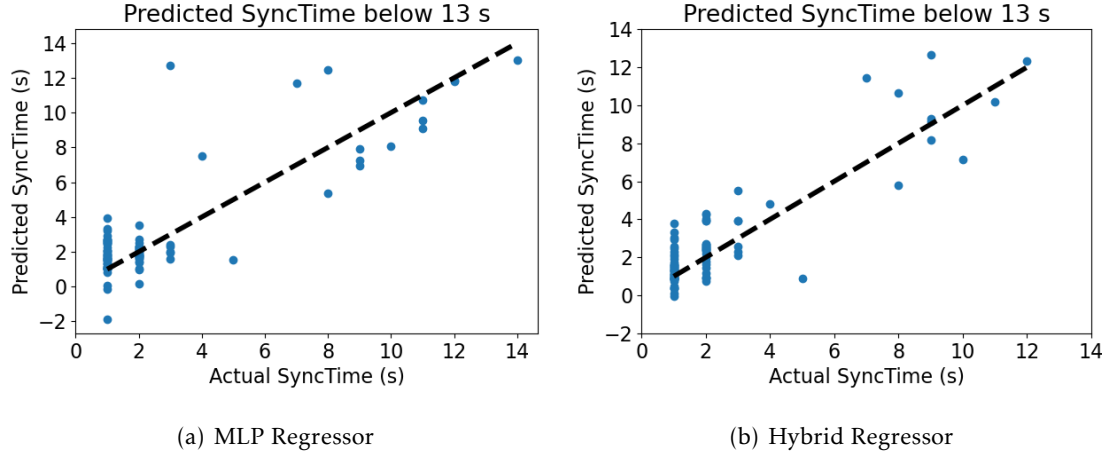


Figure 7.5: Predicted SyncTime values below 13 s

in subsection 6.2.1). Both regression models, presented in this section, do not have many predictions above the 150 s mark and do not accurately predicted any samples that have a SyncTime above 150 s.

We can observe in subfigure 7.5(a) that the MLP Regressor model presents some predicted values that are below or equal to 0 s. The presence of zeros and negative predictions implies that the regression model could not fully represent the train data (more specifically, it could not learn the lower bound of SyncTime values, which is 1 s). In turn, if we look at subfigure 7.5(b), we see that the Hybrid Regressor model does not show any negative predictions. This fact is a strong indication that the Hybrid Regressor model better fits the train data than the MLP Regressor model.

7.3 Classification vs. Regression Approaches

Figure 7.6 shows the same two subfigures views present in figure 7.5 although, this time, the only difference is that they show two additional red rectangles. These contain all predicted SyncTime values between 0.5 to 5.49 s. If we round to the nearest integer value (as explained in section 3.1, SyncTime is a discrete and positive parameter), it contains all samples between 1 and 5 s. In the classification approach, this range was considered to be the ideal range of SyncTime values (class I). We can observe that several predictions are contained inside these rectangles, which means that these samples can be classified as being part of class I. This fact is a good indication that we can use both approaches (Classification and regression) to deal with our problem. In resume, we can use one of the two regression models to predict the SyncTime value of a booking and then, classify this value according to the already pre-established classification rule decided in section 4.1 (a booking with SyncTime below or equal to 5 s is classified as I, if not then it is classified as NI). We named this new approach as Regression + Classification Approach.

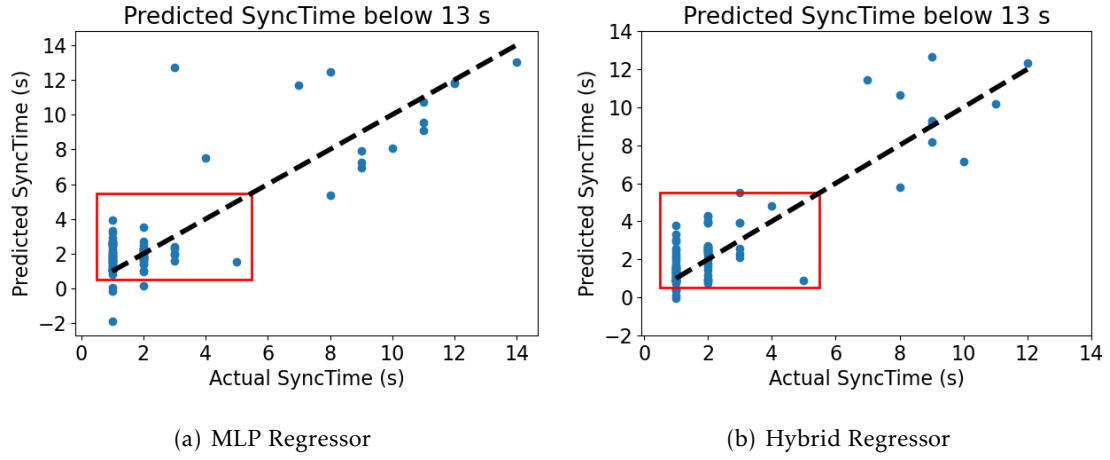


Figure 7.6: Predicted SyncTime values below 13 s with class I samples identified by red rectangles

In order to compare this new approach with the Classification approach, we classified the predictions of both models according to the pre-established classification rule and then, converted all predicted SyncTime values that are below or equal to 0 s to 1 s. By doing these, we could obtain some of the classification metrics described in subsection 2.11.3. Table 7.3 shows the value of these classification metrics for the MLP Classifier, MLP Regressor and Hybrid Regressor.

Table 7.3: Classification vs. Regression Approach Results

| Performance Metrics | MLP Classifier | MLP Regressor | Hybrid Regressor |
|---------------------|----------------|---------------|------------------|
| Accuracy | 1.0000 | 0.9797 | 1.0000 |
| F1-Score | 1.0000 | 0.9784 | 1.0000 |
| Precision | 1.0000 | 0.9714 | 1.0000 |
| Recall | 1.0000 | 0.9855 | 1.0000 |

As we can see in the table above, both the MLP Classifier and Hybrid Regressor obtained perfect results in all performance metrics compared to the other implementation. These results suggest that if we implement an MLP classifier to predict the class of a booking's SyncTime, it will give equal results compared to predicting the SyncTime value of each booking using the Hybrid Regressor model and then, classify the predicted SyncTime in two classes.

Figures 7.7 and 7.8 show the confusion matrices obtained by the MLP Regressor and Hybrid Regressor. We can see that the MLP Regressor obtained two false positives and one false negative. This is why this implementation achieved a higher value in the recall metric compared to the precision metric. This means that the MLP Regressor model almost detected all class NI samples (higher recall) in the test dataset but, could not correctly classify all class I samples. In comparison, the Hybrid Regressor did not

have any false negative or false positive which justifies the perfect results obtained in all performance metrics.

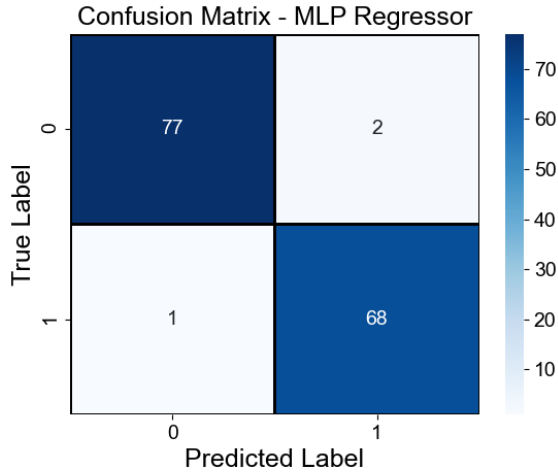


Figure 7.7: MLP Regressor Confusion Matrix

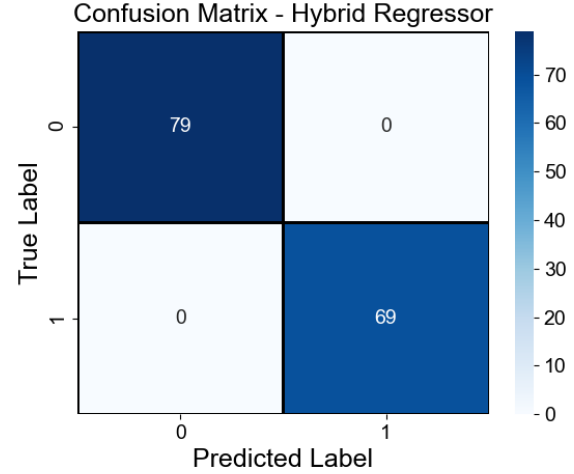


Figure 7.8: Hybrid Regressor Confusion Matrix

Table 7.4 shows a comparison made between the MLP Classifier and the Hybrid Regressor in terms of number of layers and number of units used by both implementations and the number of training epochs that were required in order to train both implementations.

Table 7.4: Comparison between the MLP Classifier and Hybrid Regressor

| Parameters | MLP Classifier | Hybrid Regressor |
|--------------------|----------------|------------------|
| N° Units | 1 | 8-8 |
| N° Training Epochs | 63 | 5358 |

We can see that the MLP classifier is composed of one unit and one hidden layer while the Hybrid Regressor uses two hidden layers with eight units in each layer. This means that the MLP Classifier achieve the same results with a much more simpler configuration. Furthermore, due to a more simpler network, the MLP Classifier has much less training epochs than the Hybrid Regressor, which means that the first has a much lower training duration than the latter.

7.4 Developed Software

One of the main contributions of this thesis was the portable prototype software that was developed, based on the results presented in the two previous sections. The main goal of this software is to aid in the assessment and development of future SRM projects deployment.

Before beginning the development process, we had to decide which ML implementation to choose of all the implementations done in this thesis, in order to integrate it in the software. After discussing the results mentioned in the previous two sections with Skyline, we chose the MLP Classifier implementation.

The software was fully developed in Python and its main feature is the ability to predict the class of a booking's SyncTime, based on a set of input parameters. The classes that were considered were the same used in this dissertation, which were class I (booking has a SyncTime below or equal to 5 s) and class NI (booking's SyncTime above 5 s). The *PyQt* v5.15 library (see [42] for more details) was used to develop the main GUI (Graphical User Interface) of the application. This library was chosen because it comes with a graphical editor, which is the Qt Designer. This editor is used to make the task of building a GUI easier by allowing users to drag-and-drop Qt Widgets (tables, textboxes, calenders and more [43]) in order to customize windows.

One requirement made by Skyline was that this application had to be portable, which means that the client can run the application in a Microsoft Windows operating system without having to install a Python compiler or any Python package. In order to make this possible, we used an application building tool called *PyInstaller* [59]. This tool allows the conversion of a Python application and all its dependencies into a stand-alone executable.

The developed software main menu is shown in figure 7.9. This menu has three options:

- **New Prediction:** Allows the user to make a prediction based on specific inputs required by the MLP Classifier. This option is the main function of this software;
- **Prediction Logs:** History of all user predictions. These predictions are listed in a table to make the visualization of all user predictions easier;
- **About:** Contains a brief description of the software features.

The implementation of the New Prediction option can be summarized by a diagram present in figure 7.10.

First, the user has to input in the application GUI the values of several required parameters. These are the DataMiner, DataMiner Agents and Index parameters:

- The DataMiner parameters are composed of all booking specific parameters plus the first subgroup of the cluster specific parameters (both are explained in section 3.2);

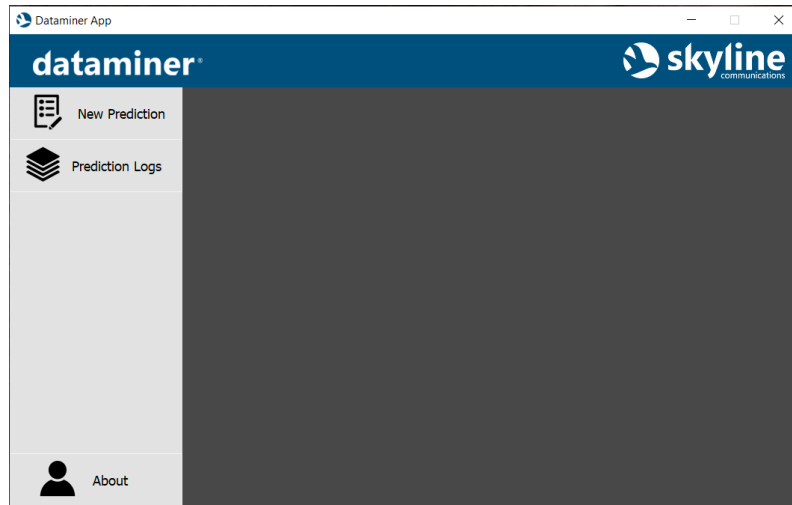


Figure 7.9: Application Main GUI

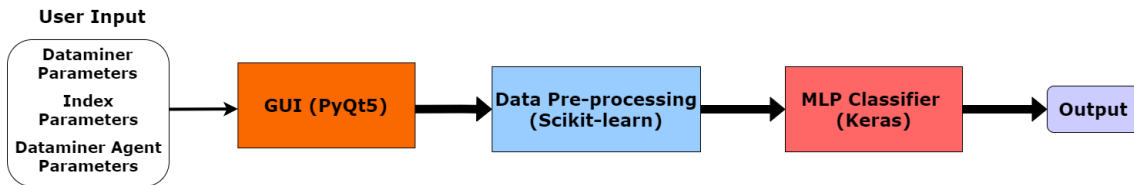


Figure 7.10: New Prediction Diagram

- The DataMiner Agent parameters are specific to each agent and, they are composed of the physical memory usage, total processor load and the ping between each agent;
- The index parameters are composed of the booking name (used only to distinguish different predictions), description (text field that can be use to describe, for example, the test or prediction that the user is trying to do), timestamp 1 and timestamp 2. These are optional but, if the user wants to import .csv files with ping data, they are required.

After the input stage, all user data is then pre-processed by a pre-implemented scaler. This was the same scaler used in the MLP implementation, described in subsection 6.2.2. We stored and loaded this scaler from a file, using Python’s Pickle module [44]. This module allows the serialization and de-serialization of a Python object structure (in this case, it is a StandardScaler object) in order to be stored or loaded from a file. The main advantage of reusing a scaler is that we do not need to fit a scaler to the input data, every time a new prediction is requested.

Next, the scaled data is fed to the MLP Classifier as input data. The MLP classifier model used in the software is a pre-trained model. After finishing the training of the MLP Classifier, we exported the resulted model into a .h5 file using the ModelCheckpoint callback function [58]. This function is available in *Keras* ML library and it is used to

export neural network models into a checkpoint file, after some interval passes. In the context of this thesis, this interval is the number of epochs which the EarlyStopping function halted the training of the MLP. The output of the MLP classifier model is a label (I or NI) that identifies the input data classification. The checkpoint file format is by default .h5 and it is used by ModelCheckpoint to store a large amount of data, which includes the MLP Classifier architecture.

Finally, we want to refer the existence of the ML_Models folder. This folder was created so that the user could change both the MLP model and the scaler used in this software. The user only has to replace the contents of said folder with exported files of other scalers and/or other neural network models.

CONCLUSION & FUTURE WORK

8.1 Conclusion

Throughout this dissertation, we carried out a detailed analysis of the state of art of machine learning methods. This study was done in order to develop a delay estimator/classifier module based on machine learning methods. This module is capable of predicting/classifying the SyncTime, based on several booking specific parameters such as, the number of resources of each booking or the number of concurrent booking. Furthermore, several cluster specific parameters are also considered like the total processor load and the physical memory usage per DataMiner agent. In order to implement this estimator/classifier, two approaches were considered: Classification of the SyncTime based on two classes (Ideal or Not Ideal) and prediction of the SyncTime value.

In chapter 2, we presented a brief introduction to machine learning and made a review on the state of art methods that can be used for classification and prediction problems.

As described in chapter 3, the case study considered in this thesis was a cluster composed of three DataMiner Agents. In the data collection process, 13 cluster setups were defined in order to capture the cluster in different network conditions (e.g, high processor load in all agents, low processor load in all agents, high memory usage in all agents). From this process, we acquired the dataset used in the implementation of the classification and regression approaches.

As described in chapter 4, we analyzed the obtained dataset using three different methods: CDF, t-SNE and Correlation matrix. The CDF allowed us to discover the frequency of samples per SyncTime value in the dataset. Furthermore, the CDF aided us in selecting the threshold between the two classes (class I and NI) used in the classification

approach. Based on the CDF results, on Skyline's business model and on what the clients of said company considered to be an Ideal SyncTime value, the threshold selected was 5 s. After choosing this value, we discovered that the obtained dataset was slightly unbalanced dataset because, it contained more samples from class NI than from class I. Then, to better understand the data acquired, we visualized it using the t-SNE method. Finally, the correlation matrices obtained from all setups showed us the linear contribution of all input variables in the SyncTime of a booking. From these matrices, we discovered which parameters had a bigger impact on the SyncTime of a booking.

As described in chapters 5 and 6, we implemented some of the classification and regression ML algorithms presented on chapter 2. The classification methods implemented were: MLP, SVM and Random forest. The regression methods implemented were: MLP Regressor and Hybrid Regressor (MLP + Linear Regression).

As described in chapter 7, we analyzed the results obtained by the best implementations of the classification and regression approaches. From this analysis, we reached some important conclusions:

- Considering the classification approach, the implementation that obtained the best results was the MLP Classifier;
- In the regression approach, the Hybrid Regressor model obtained a lower RMSE and MAE value than the MLP Regressor. Furthermore, the Hybrid Regressor model only showed positive predicted values, unlike the MLP Regressor model. This fact proves that the Hybrid Regressor model better fits the training data;
- Finally, to compare the classification approach with the regression approach, we decided to classify the predicted values of the MLP Regressor and Hybrid Regressor models. After this, we could obtain some of the classification metrics considered in the classification approach such, as the F1-score, Accuracy, Precision, and Recall. Comparing the MLP Regressor and Hybrid Regressor with the MLP Classifier, we observed that the MLP Classifier and the Hybrid Regressor obtained better results compared to the MLP Regressor. Then, we concluded that the Hybrid Regressor is a more complex neural network than the MLP Classifier. This is because the first one has one more hidden layer than the latter. Furthermore, the Hybrid Regressor uses eight units per layer while the latter only uses one unit. Finally, the Hybrid Regressor has much more training epochs than the MLP Classifier, which means that the first has a much higher training duration than the latter.

The last objective of this dissertation is the development of a portable desktop application, which uses the best estimator/classifier implemented to predict/classify the SyncTime of a booking, based on several input values given by the user. After presenting

the results obtained in this dissertation to Skyline, the implementation selected to be integrated in this prototype was the MLP Classifier (see the implementation process of this MLP in subsection 6.2.2). The developed prototype and its principal functionalities were shown in the last section of chapter 7.

Finally, we can say that the three main contributions of this thesis are:

- The study presented in this dissertation allowed us to discover which parameters had more impact in the SyncTime of a booking.
- We selected the best approaches that achieve the best results in all evaluation metrics considered;
- We developed a prototype application which classifies the SyncTime of a booking based on a set of input parameters. This classification is done by a pre-trained MLP. The goal of this software is to aid in the assessment and development of future SRM projects deployment.

8.2 Future Work

Some work can be done in the future to further improve the study presented in this dissertation. This work can be summarized as follows:

- In this dissertation, we were limited to a case study of a cluster composed of three DataMiner Agents. As future work, we want to expand the current work by considering more complex clusters, composed of more than three DataMiner Agents;
- Implementation and performance testing of PCA, Autoencoder, and LSTM. These methods were described in chapter 2 but were not taken into account in this dissertation due to limitations of time. LSTM will be primarily used to optimize the regression approach by using the cluster history as the input dataset. Both the Autoencoder and PCA will be used to optimize the results of both approaches. As seen in [60] and [10], these methods can improve the overall performance of a classifier/estimator by compressing the input dataset before feeding it to the classifier/estimator.

BIBLIOGRAPHY

- [1] C. C. Aggarwal. “Data Mining: The Textbook”. In: Springer Publishing Company, Incorporated, 2015. Chap. 10, p. 328. ISBN: 3319141414 (cit. on p. 13).
- [2] C. C. Aggarwal. *Neural Networks and Deep Learning*. Springer, 2018 (cit. on pp. 16, 17).
- [3] C. Albon. *Machine Learning with Python Cookbook: Practical Solutions from Preprocessing to Deep Learning*. O’Reilly Media, Inc., 2018. ISBN: 1491989386 (cit. on pp. 7, 8).
- [4] T. API. *tf.keras.callbacks.EarlyStopping*. Available online at https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping. Accessed: 2021-08-25 (cit. on p. 55).
- [5] A. Azzouni and G. Pujolle. “A Long Short-Term Memory Recurrent Neural Network Framework for Network Traffic Matrix Prediction”. In: *CoRR* abs/1705.05690 (2017). arXiv: 1705.05690. URL: <http://arxiv.org/abs/1705.05690> (cit. on p. 20).
- [6] N. Barls. *Overfitting vs Underfitting in Machine Learning – Everything You Need to Know*. Available online at <https://neptune.ai/blog/overfitting-vs-underfitting-in-machine-learning>. Accessed: 2021-08-26 (cit. on p. 55).
- [7] J. Brownlee. *Bagging and Random Forest Ensemble Algorithms for Machine Learning*. Available online at <https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html>. Accessed: 2021-09-15 (cit. on p. 52).
- [8] J. Brownlee. *Difference Between Classification and Regression in Machine Learning*. Available online at <https://machinelearningmastery.com/classification-versus-regression-in-machine-learning/>. Accessed: 2021-10-31 (cit. on p. 5).
- [9] J. Brownlee. *How to use Learning Curves to Diagnose Machine Learning Model Performance*. Blog post available online at <https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-performance/>. Accessed: 2021-08-23 (cit. on pp. 54, 57).

-
- [10] F. O. Catak and A. Mustacoglu. “Distributed denial of service attack detection using autoencoder and deep neural networks”. In: *Journal of Intelligent & Fuzzy Systems* 37 (July 2019), pp. 1–11. DOI: [10.3233/JIFS-190159](https://doi.org/10.3233/JIFS-190159) (cit. on pp. 18, 19, 79).
 - [11] B. J. Cavalcanti et al. “A hybrid path loss prediction model based on artificial neural networks using empirical models for LTE and LTE-A at 800 MHz and 2600 MHz”. In: *Journal of Microwaves, Optoelectronics and Electromagnetic Applications* 16.3 (2017), pp. 708–722. ISSN: 21791074. DOI: [10.1590/2179-10742017v16i3925](https://doi.org/10.1590/2179-10742017v16i3925) (cit. on pp. 23, 24).
 - [12] E. Charniak. “Introduction to Deep Learning”. In: The MIT Press, 2018. Chap. World Embeddings and Recurrent NNs, pp. 89–92 (cit. on p. 20).
 - [13] Y. Dai and G. Wang. “Analyzing Tongue Images Using a Conceptual Alignment Deep Autoencoder”. In: *IEEE Access* 6 (2018), pp. 5962–5972. DOI: [10.1109/ACCESS.2017.2788849](https://doi.org/10.1109/ACCESS.2017.2788849) (cit. on p. 17).
 - [14] A. Dertat. *Applied Deep Learning - Part 3: Autoencoders*. Available Online: <https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798>. Accessed: 2020-10-24. 2017 (cit. on p. 18).
 - [15] scikit-learn developers. *sklearn.svm.SVC*. Available Online at <https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html>. Accessed: 2020-01-18. Jan. 2021 (cit. on p. 50).
 - [16] scikit-learn developers. *sklearn.ensemble.RandomForestClassifier*. Available Online at <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html>. Accessed: 2021-01-17 (cit. on p. 52).
 - [17] V. Dutta et al. “Hybrid model for improving the classification effectiveness of network intrusion detection”. In: *Advances in Intelligent Systems and Computing* 1267 AISC (2021), pp. 405–414. ISSN: 21945365 (cit. on p. 22).
 - [18] I. C. Education. *Recurrent Neural Networks*. Available online at <https://www.ibm.com/cloud/learn/recurrent-neural-networks>. Accessed: 2021-10-31 (cit. on p. 20).
 - [19] N. Farnaaz and J. Akhil. “Random Forest Modeling for Network Intrusion Detection System”. In: *Procedia Computer Science* 89 (Dec. 2016), pp. 213–217. DOI: [10.1016/j.procs.2016.06.047](https://doi.org/10.1016/j.procs.2016.06.047) (cit. on p. 10).
 - [20] M. B. Fraj. *In Depth: Parameter tuning for Random Forest*. Available online at <https://medium.com/all-things-ai/in-depth-parameter-tuning-for-random-forest-d67bb7e920d>. Accessed: 2021-01-18. Dec. 2017 (cit. on p. 52).
 - [21] M. B. Fraj. *In Depth: Parameter tuning for SVC*. Available online at <https://medium.com/all-things-ai/in-depth-parameter-tuning-for-svc-758215394769>. Accessed: 2021-01-18. Jan. 2018 (cit. on p. 51).

- [22] H. He and Y. Ma. “Imbalanced Learning: Foundations, Algorithms, and Applications”. In: 1st ed. Wiley-IEEE Press, July 2013. Chap. 3 (cit. on p. 42).
- [23] B. Holländer. *Autoencoders: Overview of Research and Applications*. Available Online at <https://towardsdatascience.com/autoencoders-overview-of-research-and-applications-86135f7c0d35>. Accessed: 2020-11-04. Oct. 2020 (cit. on pp. 17, 18).
- [24] M. Iqbal and Z. Yan. “SUPERVISED MACHINE LEARNING APPROACHES: A SURVEY”. In: *International Journal of Soft Computing* 5 (Apr. 2015), pp. 946–952. DOI: [10.21917/ijsc.2015.0133](https://doi.org/10.21917/ijsc.2015.0133) (cit. on p. 6).
- [25] Javapoint. *Introduction to Semi-Supervised Learning*. Available online at <https://www.javatpoint.com/semi-supervised-learning>. Accessed: 2021-10-31 (cit. on p. 6).
- [26] M. K. Jiawei Han and J. Pei. “Data Mining: Concepts and Techniques”. In: ed. by M. Kaufmann. Third edition. Elsevier, 2012. Chap. 9, pp. 408–415. DOI: <https://doi.org/10.1016/C2009-0-61819-5> (cit. on pp. 11, 12).
- [27] J. Jordan. *Introduction to autoencoders*. Available Online at <https://www.jeremyjordan.me/autoencoders>. Accessed: 2020-10-24. Mar. 2018 (cit. on pp. 16–18).
- [28] W. Koehrsen. *Hyperparameter Tuning the Random Forest in Python*. Available online at <https://towardsdatascience.com/hyperparameter-tuning-the-random-forest-in-python-using-scikit-learn-28d2aa77dd74>. Accessed: 2021-01-18. 2018 (cit. on p. 52).
- [29] M. Kuhn and K. Johnson. “Applied Predictive Modeling with Applications in R”. In: vol. 26. Springer, 2013. Chap. 4, pp. 72–73. ISBN: 9781461468486 (cit. on p. 52).
- [30] A. M. Kumar. *C and Gamma in SVM*. Available online at <https://medium.com/@myselfaman12345/c-and-gamma-in-svm-e6cee48626be>. Accessed: 2021-11-02. Dec. 2018 (cit. on p. 50).
- [31] T. Le et al. “Machine Learning Methods for Reliable Resource Provisioning in Edge-Cloud Computing: A Survey”. In: *ACM Computing Surveys* 52 (Sept. 2019), pp. 1–39. DOI: [10.1145/3341145](https://doi.org/10.1145/3341145) (cit. on p. 5).
- [32] C. Liu. *SVM-Hyper-parameter-Tuning-using-GridSearchCV*. Available online at <https://github.com/clareyan/SVM-Hyper-parameter-Tuning-using-GridSearchCV>. Accessed: 2021-11-03 (cit. on p. 51).
- [33] J. Lv, M. Wei, and Y. Yu. “A Container Scheduling Strategy Based on Machine Learning in Microservice Architecture”. In: *2019 IEEE International Conference on Services Computing(SCC)*. 2019, pp. 65–71. DOI: [10.1109/SCC.2019.00023](https://doi.org/10.1109/SCC.2019.00023) (cit. on p. 10).

-
- [34] Microsoft. *CORREL function*. Available online at <https://support.microsoft.com/en-us/office/correl-function-995dcef7-0c0a-4bed-a3fb-239d7b68ca92>. Accessed: 2021-08-14 (cit. on p. 46).
- [35] B. Mohammed et al. "Failure prediction using machine learning in a virtualized HPC system and application". In: *Cluster Computing* 22 (June 2019). DOI: [10.1007/s10586-019-02917-1](https://doi.org/10.1007/s10586-019-02917-1) (cit. on p. 12).
- [36] A. Mustafa and Y. Swamy. "Web Service classification using Multi-Layer Perceptron optimized with Tabu search". In: *Souvenir of the 2015 IEEE International Advance Computing Conference, IACC 2015* (July 2015), pp. 290–294. DOI: [10.1109/IADCC.2015.7154716](https://doi.org/10.1109/IADCC.2015.7154716) (cit. on p. 16).
- [37] S. Narkhede. *Understanding Confusion Matrix*. Available online at <https://towardsdatascience.com/understanding-confusion-matrix-a9ad42dcfd62>. Accessed: 2021-11-01. May 2018 (cit. on pp. 26, 28).
- [38] H. Nguyen et al. "ESNemle: an Echo State Network-based ensemble for workload prediction and resource allocation of Web applications in the cloud". In: *The Journal of Supercomputing* 75 (Apr. 2019). DOI: [10.1007/s11227-019-02851-4](https://doi.org/10.1007/s11227-019-02851-4) (cit. on p. 8).
- [39] Z. Noshad et al. "Fault Detection in Wireless Sensor Networks through the Random Forest Classifier". In: *Sensors* 19 (Apr. 2019), p. 1568. DOI: [10.3390/s19071568](https://doi.org/10.3390/s19071568) (cit. on p. 12).
- [40] C. Nwankpa et al. "Activation Functions: Comparison of trends in Practice and Research for Deep Learning". In: (2018), pp. 1–20. arXiv: [1811.03378](https://arxiv.org/abs/1811.03378). URL: <http://arxiv.org/abs/1811.03378> (cit. on p. 13).
- [41] J. Patterson and A. Gibson. "Deep Learning: A Practitioner's Approach". In: 1st. O'Reilly Media, Inc., 2017. Chap. 2, pp. 65–71. ISBN: 1491914254 (cit. on p. 13).
- [42] PyQT. *PyQt5 Reference Guide*. Available online at <https://www.riverbankcomputing.com/static/Docs/PyQt5/index.html>. Accessed: 2021-10-04 (cit. on p. 74).
- [43] PyQT. *Widgets Classes*. Available online at <https://doc.qt.io/qt-5/widget-classes.html>. Accessed: 2021-10-04 (cit. on p. 74).
- [44] Python. *pickle — Python object serialization*. Available online at <https://docs.python.org/3.8/library/pickle.html>. Accessed: 2021-10-04 (cit. on p. 75).
- [45] G.-J. Qi and J. Luo. "Small Data Challenges in Big Data Era: A Survey of Recent Progress on Unsupervised and Semi-Supervised Methods". In: *IEEE transactions on pattern analysis and machine intelligence* (2020) (cit. on p. 6).
- [46] J. Rahman and P. Lama. "Predicting the End-to-End Tail Latency of Containerized Microservices in the Cloud". In: *2019 IEEE International Conference on Cloud Engineering (IC2E)*. 2019, pp. 200–210. DOI: [10.1109/IC2E.2019.00034](https://doi.org/10.1109/IC2E.2019.00034) (cit. on p. 15).

- [47] M. Russinovich. *CpuStres v2.0*. Blog post available at <https://docs.microsoft.com/en-us/sysinternals/downloads/testlimit>. Accessed: 2021-07-17. 2018 (cit. on p. 35).
- [48] M. Said Elsayed et al. “Network Anomaly Detection Using LSTM Based Autoencoder”. In: New York, NY, USA: Association for Computing Machinery, 2020, pp. 37–45. ISBN: 9781450381208. URL: <https://doi.org/10.1145/3416013.3426457> (cit. on p. 21).
- [49] M. Sakurada and T. Yairi. “Anomaly Detection Using Autoencoders with Nonlinear Dimensionality Reduction”. In: New York, NY, USA: Association for Computing Machinery, 2014, pp. 4–11. ISBN: 9781450331593. DOI: [10.1145/2689746.2689747](https://doi.org/10.1145/2689746.2689747). URL: <https://doi.org/10.1145/2689746.2689747> (cit. on p. 19).
- [50] Sklearn. *sklearn.linear_model.LinearRegression*. Available online at https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html. Accessed: 2021-09-22 (cit. on p. 64).
- [51] Sklearn. *sklearn.model_selection.KFold*. Available online at https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html. Accessed: 2021-08-20 (cit. on p. 25).
- [52] Sklearn. *sklearn.preprocessing.MinMaxScaler*. Available online at <https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html>. Accessed: 2021-08-29 (cit. on p. 59).
- [53] Sklearn. *sklearn.preprocessing.StandardScaler*. Available online at <https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html>. Accessed: 2021-08-18 (cit. on p. 50).
- [54] *sklearn.manifold.TSNE*. Available online at <https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html>. Accessed: 2021-01-22. Jan. 2021 (cit. on p. 43).
- [55] *sklearn.model_selection.train_test_split*. Available online at https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html. Accessed: 2021-02-02. Feb. 2021 (cit. on pp. 49, 60).
- [56] *sklearn.modelselection.RandomizedSearchCV*. Available online at https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html. Accessed: 2021-01-18. Jan. 2021 (cit. on pp. 25, 51, 52).
- [57] C. Tao. *clumsy v0.2*. Available online at <https://jagt.github.io/clumsy/index.html>. Accessed: 2021-07-18 (cit. on p. 35).
- [58] K. Team. *ModelCheckpoint*. Available online at https://keras.io/api/callbacks/model_checkpoint/. Accessed: 2021-10-04 (cit. on p. 75).
- [59] P. D. Team. *PyInstaller*. Github repository: <https://github.com/pyinstaller/pyinstaller>. Accessed: 2021-10-06 (cit. on p. 74).

-
- [60] S. Thaseen. “Improving Accuracy of Intrusion Detection Model Using PCA and optimized SVM”. In: *Journal of Computing and Information Technology* 24 (June 2016), pp. 133–148. DOI: [10.20532/cit.2016.1002701](https://doi.org/10.20532/cit.2016.1002701) (cit. on pp. 8, 79).
 - [61] A. J. Thomas et al. “Two hidden layers are usually better than one”. In: *Communications in Computer and Information Science* 744 (2017), pp. 279–290. ISSN: 18650929. DOI: [10.1007/978-3-319-65172-9_24](https://doi.org/10.1007/978-3-319-65172-9_24) (cit. on p. 15).
 - [62] K. Vijayshinva. *Tools To Simulate CPU / Memory / Disk Load*. Blog post available at <https://docs.microsoft.com/pt-pt/archive/blogs/vijaysk/tools-to-simulate-cpu-memory-disk-load>. Accessed: 2021-07-17. 2012 (cit. on p. 35).
 - [63] S. Weiran. *Hyper Parameter Tuning with Randomised Grid Search*. Available online at <https://towardsdatascience.com/hyper-parameter-tuning-with-randomised-grid-search-54f865d27926>. Accessed: 2021-11-03. Sept. 2019 (cit. on p. 51).
 - [64] A. Ye. *Supervised Learning, But A Lot Better: Semi-Supervised Learning*. Available online at <https://towardsdatascience.com/supervised-learning-but-a-lot-better-semi-supervised-learning-a42dff534781>. Accessed: 2021-10-31 (cit. on p. 6).
 - [65] X. Ying. “An Overview of Overfitting and its Solutions”. In: *Journal of Physics: Conference Series* 1168.2 (2019). ISSN: 17426596. DOI: [10.1088/1742-6596/1168/2/022022](https://doi.org/10.1088/1742-6596/1168/2/022022) (cit. on p. 55).
 - [66] T. Yiu. *Understanding Random Forest. How the Algorithm Works and Why it Is So Effective*. Online. Available at <https://towardsdatascience.com/understanding-random-forest-58381e0602d2>. Accessed: 2021-02-11. June 12, 2019 (cit. on pp. 8–10).
 - [67] S. Yıldırım. *Hyperparameter Tuning for Support Vector Machines — C and Gamma Parameters*. Available online at <https://towardsdatascience.com/hyperparameter-tuning-for-support-vector-machines-c-and-gamma-parameters-6a5097416167>. Accessed: 2021-11-03. May 2020 (cit. on p. 50).
 - [68] P. Yosifovich. *CpuStres v2.0*. Blog post available at <https://docs.microsoft.com/en-us/sysinternals/downloads/cpustres>. Accessed: 2021-07-17. 2018 (cit. on p. 34).
 - [69] Q. Zhang et al. “Prediction of Sea Surface Temperature using Long Short-Term Memory”. In: *CoRR* abs/1705.06861 (2017). arXiv: [1705.06861](https://arxiv.org/abs/1705.06861). URL: <http://arxiv.org/abs/1705.06861> (cit. on p. 21).
 - [70] A. Zheng and A. Casari. *Feature engineering for machine learning*. September. 2018, p. 218. ISBN: 978-1491953242 (cit. on p. 7).

