UNIVERSIDAD POLITÉCNICA DE
MADRID

ESCUELA TÉCNICA SUPERIOR DE
INGENIEROS DE TELECOMUNICACIÓN

**Machine Learning based predictive
modelling and simulation of network
failure events**

TESIS DOCTORAL

JAVIER ANDIÓN JIMÉNEZ
Ingeniero de Telecomunicación

Septiembre, 2020

Abstract

Proactive failure management can produce high cost saves for companies. Machine Learning has proven to be an excellent tool for creating fault prediction models, and there are numerous scientific publications with examples of applications in different domains. In the area of information technologies, its use in the management of distributed systems stands out. However, these failure prediction publications with Machine Learning typically do not present a detailed description of data pre-processing, or in other cases, they are specific to the application scenario. These facts difficult to reuse the results of the literature or its application in new scenarios.

This thesis's first contribution is the proposal of a generic methodology for creating prediction models with Machine Learning algorithms. This contribution is based on a research line in which several articles have been published in the last five years. This methodology aims to propose a guide that starts with the raw data and ends with prediction models adapted to the problem and validated correctly. The second contribution is the architectural proposal of a simulator based on the use of prediction models. Prediction models can detect patterns of activity in the monitored system that trigger failures. This capability can be used in reverse to simulate the occurrence of failures when certain conditions occur.

Both contributions are validated using a dataset provided by a Spanish bank. The dataset consists of a network monitoring system's logs working on a production network for nine months. Contributions are designed as scalable and extensible solutions that can be applied to larger or more complex scenarios.

Resumen

La gestión proactiva de fallos permite ahorrar costes independientemente del dominio de aplicación. El Machine Learning ha probado ser una excelente herramienta para crear modelos de predicción de fallos y existen numerosas publicaciones científicas con ejemplos de aplicación a distintos dominios. En el área de las tecnologías de información destaca su uso en la gestión de sistemas distribuidos. Sin embargo, estas publicaciones de predicción de fallos con Machine Learning no suelen presentar un proceso detallado del tratamiento de los datos que se lleva a cabo, o en otros casos, pueden ser específicas del escenario de trabajo. Estos hechos dificultan la reutilización de los resultados de la literatura o su aplicación en nuevos escenarios. La primera contribución de este trabajo es la propuesta de una metodología genérica para la creación de modelos de predicción con algoritmos de Machine Learning. Esta contribución se sustenta en una línea de investigación de varios años en la que se han generado y publicado en revistas varios artículos. El objetivo de esta metodología es proponer una guía que comience con los datos en bruto y finalice con unos modelos de predicción adaptados al problema y validados correctamente. La segunda contribución es la propuesta de arquitectura de un simulador basada en la utilización de los modelos de predicción. Los modelos de predicción son capaces de detectar patrones de actividad en el sistema monitorizado que desencadenan la aparición de fallos, esta capacidad puede emplearse de forma inversa para simular la aparición de fallos cuando se den las condiciones adecuadas. Ambas contribuciones se validan utilizando un conjunto de datos cedido por un banco español. Este conjunto consiste en los registros de un sistema de monitorización de red que trabaja sobre una subred de producción durante nueve meses. Las contribuciones se diseñan como soluciones escalables y extensibles que puedan ser aplicados a escenarios de mayor dimensión o complejidad.

Acknowledgments

Empiezo estos agradecimientos, como no puede ser de otra forma, dando las gracias a mi tutor. Gracias Juan Carlos por tu guía, tu confianza y sobre todo por darme la oportunidad de hacer este doctorado. Estos años han sido una experiencia que me ha cambiado y me alegro de haber emprendido este camino. Félix, gracias por aguantarme los últimos meses de redacción de esta tesis y todos los consejos que me has dado. Gracias también por la estancia en Londres, pude conocer gente interesante y observar la investigación en grupo desde un nuevo punto de vista. Hugo, gracias por no dejar de recordarme que tenía que terminar esta tesis. Gracias July, sin tu ayuda seguro que habría acabado perdido en algún trámite por el rectorado. José Luis, gracias por darme la increíble oportunidad de conocer Colombia y por el recibimiento que me disteis los meses que estuve allí. José Manuel, mi revisor más exigente y mi compañero de trabajo durante casi todo el doctorado. Gracias, esto solo ha sido posible con tu ayuda.

En cuanto a mis amigos (además de los que ya he nombrado), tengo la inusual suerte de contar con un gran número de personas a las que os considero mis amigos. Si intento enumeraros a todos, es muy probable que me olvide de alguno de vosotros. Así que prefiero que te sientas identificado/a leyendo estas palabras. Aunque te parezca que no tienes relación con lo que he hecho en este doctorado, tu compañía es ha sido un hecho crítico que me ha permitido llegar a desarrollar todas las ideas que cuento en esta tesis. Te doy las gracias por todos esos momentos, charlando, viajando, tomando cervezas, jugando al rol cara a cara o por internet, las innumerables partidas de juegos de mesa, cocinando con artilugios extraños, intentando recetas que siempre salen mal o simplemente estando ahí; en resumen: Gracias por tu amistad.

Finalmente quiero dar las gracias a mis padres, hermano y al resto familia por vuestro apoyo. Creo que no siempre he sido capaz de haceros entender o explicar a qué es eso de hacer un doctorado o a que me dedicaba, y aun así siempre he contado con vuestros ánimos incondicionales. Este trabajo os lo dedico a todos vosotros.

Take it
E.M. 19

Contents

1	Introduction	1
1.1	Methodology	3
1.1.1	Knowledge Discovery in Databases	3
1.1.2	Design Science Paradigm	5
1.1.3	Followed Methodology	7
1.2	Document structure	8
2	State of the Art	9
2.1	Machine Learning Failure Prediction	12
2.1.1	Data windowing	13
2.1.2	Unbalanced data	15
2.1.3	Evaluation metrics	15
2.1.4	Algorithms	17
2.1.5	Data availability	17
2.2	Simulation	18
2.3	Discussion	20
2.4	Chapter summary	22
3	Objectives	24
3.1	Objectives	24
3.1.1	Objective 1: propose a general, documented, and complete methodology to create Machine Learning prediction models for failure prediction	25
3.1.2	Objective 2: design the architecture of a discrete failure simulator based on prediction models	25
3.2	Requirements	25
3.3	Processing environment	27
3.3.1	Environment A: cluster machine	28
3.3.2	Environment B: laptop	28
3.4	Chapter summary	29

4	Scenario description	30
4.1	Background	30
4.2	Event interpretation	31
4.3	Dataset description	34
4.4	Data cleaning and preprocessing	38
4.5	Chapter summary	41
5	Generic methodology to generate ML prediction models from discrete events dataset	42
5.1	Prediction model definition	42
5.2	Data windowing techniques	43
5.3	Methods to evaluate prediction models	44
5.4	Model specification	48
5.5	Methodology proposal	51
5.6	Methodology in detail	54
5.6.1	Classic windowing	54
5.6.2	Instantaneous prediction	58
5.6.3	Observation window size optimization	62
5.6.4	Model selection	67
5.6.5	Feature selection	73
5.6.6	Sampling methods	78
5.7	Conclusions	81
5.8	Chapter summary	83
6	Design and implementation of a discrete network event simulator	85
6.1	Design	85
6.1.1	Agent based architecture	86
6.1.2	Agent communication	87
6.1.3	Event queue agent	88
6.1.4	Output manager agent	89
6.1.5	Model agent	90
6.2	Simulation models	90
6.2.1	Prediction models	91
6.2.2	Regression models	94
6.3	Simulation workflow	98
6.3.1	Startup and set up	98
6.3.2	Model trainer	99
6.3.3	Messages	99
6.3.4	Simulation end	99
6.3.5	Working modes	100
6.4	Chapter summary	103

7	Validation	105
7.1	Prediction models validation	105
7.2	Simulator validation	109
7.3	Chapter summary	114
8	Conclusions	115
8.1	Objectives	115
8.2	Discussion	116
8.3	Future work	118
A	Tools and software	119
A.1	Programming Languages	119
A.1.1	Scala	119
A.1.2	Python	120
A.2	Software frameworks	121
A.2.1	Spark	121
A.2.2	Pandas	122
A.2.3	Scikit Learn	123
A.2.4	Akka	124
A.2.5	Other libraries and modules	125
A.3	Environment Tools	125
A.3.1	Hadoop FS	125
A.3.2	MLFlow	126
A.3.3	Jupyter notebook	126
A.3.4	Zeppelin	127
B	Machine Learning algorithms and Data Processing Techniques	128
B.1	Machine Learning techniques	128
B.1.1	Classification models	128
B.1.2	Overfitting	129
B.1.3	Model validation through K-Folds	129
B.2	Machine Learning algorithms	131
B.2.1	Logistic Regression	131
B.2.2	Support Vector Machine	132
B.2.3	Naive Bayes	133
B.2.4	Decision Tree	133
B.2.5	Random Forest	134
B.2.6	Gradient Boosted Tree	136
B.3	Data Processing techniques	137
B.3.1	KDE	137
B.4	Agent Computing	137

C Detailed results	138
C.1 Scenario	138
C.2 Methodology	140
C.2.1 Step 1	140
C.2.2 Step 2	140
C.2.3 Step 3	142
C.2.4 Step 4	143
C.2.5 Step 5	144
Bibliography	146

List of Figures

1.1	Knowledge Discovery in Databases process diagram	4
1.2	Framework for Design Science in information systems	6
2.1	Growth of the number of publications in Machine Learning predic- tion since 1980	13
2.2	Online Prediction schema. Figure extracted from [25]	14
4.1	Records distribution by machine	32
4.2	Records distribution by machine type	32
4.3	Records distribution by message code	33
4.4	Records distribution by Event	34
4.5	Daily envent distribution over the monitoring period	35
4.6	Filtered event daily aggregation distribution	35
4.7	Boxplot of event distribution by weekday	36
4.8	Filtered event daily distribution over the monitoring period	37
4.9	Event frequencies before and after the filter	38
4.10	Histogram of event's frequencies	39
5.1	Observation-Prediction window schema	43
5.2	Prediction windows overlapping	48
5.3	Methodology steps	54
5.4	Classic windowing schema	56
5.5	Step 0: model scores	57
5.6	Step 0: the ratio of positive samples	58
5.7	Step 1: positive samples ratio	60
5.8	Step 1: models score distribution	60
5.9	Step 1: relation between models' scores and samples	61
5.10	Step 1: Boxplot and violin plot model scores	62
5.11	Step 2: Window optimization raw models scores	64
5.12	Step 2: Window optimization filtered models scores	65
5.13	Boxplots of the model variables distributions	66
5.14	Step 2: Model scores evolution	67

5.15	Step 3: Random Search iterations	68
5.16	Step 3: logistic regression results	70
5.17	Hyperparameters Random Search results	71
5.18	Step 3: Model scores evolution	72
5.19	Boxplot of the model variables distributions	74
5.20	Evaluation of Decision Tree model of event #8 depending on the number of features selected	76
5.21	Distribution of the number of features selected	77
5.22	Step 4: Model scores evolution	78
5.23	Step 5: Boxplot model scores	81
5.24	Step 5: Model scores evolution	81
5.25	Scores distributions	83
6.1	Simulator architecture	87
6.2	Agent communication	88
6.3	Prediction models F-score	91
6.4	Prediction models F-score filtered	92
6.5	Prediction models agent workflow	93
6.6	Time differences between event occurrences	94
6.7	Time differences between event occurrences with KDE	95
6.8	CDF of time differences for sample events	97
6.9	Regression models agent workflow	98
6.10	Regression model agent response time	100
6.11	Prediction model agent response time	101
6.12	General response time	102
7.1	Number of sessions versus event frequency	108
7.2	Number of True Positives and False Negative sessions of each event	108
7.3	Number of True Positives, False Negative and False Positive sessions of each event	109
7.4	Simulation output stabilization	110
7.5	Daily events for original and synthetic datasets	111
7.6	Event counts for original and synthetic datasets	112
7.7	Event counts for original and regression only synthetic datasets . .	113
7.8	Daily events for original and regression only synthetic datasets . . .	113
A.1	Scala language logo	119
A.2	Python language logo	120
A.3	Apache Spark logo	121
A.4	Pandas logo	122
A.5	Scikit Learn logo	123

A.6	Akka logo	124
A.7	Apache Hadoop logo	125
A.8	Databricks MLFlow logo	126
A.9	Jupyter logo	126
A.10	Apache Zeppelin	127
B.1	K Folds cross validation	130

List of Tables

2.1	Comparing addressed points in published works on failure prediction	20
4.1	Event samples summary	40
5.1	Confusion matrix	45
5.2	Algorithm hyperparameters configurations	69
5.3	Step 3: detailed view of different algorithm model scores	72
5.4	Step 3: Selected algorithm count	72
5.5	Sampling technique selected	80
5.6	Scores evolution	82
5.7	Computation time summary	83
6.1	Simulation duration in batch mode	103
C.1	Event frequencies	139
C.2	F-Score results of step 1	140
C.3	Genetic Algorithm optimization results in step 2	141
C.5	Step 3: model selection F-score	142
C.7	Step 3: model selection times	143
C.8	Step 4: number of selected features	144
C.9	Step 5: sampling algorithms and final samples	145

Chapter 1

Introduction

Data analytics is a powerful tool that helps individuals, companies, and industries alike to make decisions and understand the operation of the systems we deal with, like devices, computer networks, or just software. Data analytics is about extracting information that is hidden at first glance. For this purpose, it initially relied solely on statistics, but nowadays, other disciplines such as Machine Learning (ML) or visualization techniques make it possible to create models and obtain insights that would only be unattainable with statistics. Today's data collection presence is so decisive that we have changed from talking about the age of information to the age of analytics [1]. Although it has been a long time since companies realized the value of the data and virtually any imaginable device is prepared to collect and record information about its activity, many challenges in the area of analysis of this collected data are still present. One of the best known features of this type of data collection is the volume handled, which has resulted in Big Data technologies. Big data is the set of issues and technologies that define and try to solve the challenges of working with large volumes of data. In addition to the volume being worked, it should also be considered the speed at which data is generated. It is not enough to be able to handle gigabytes or terabytes of data, but it must be done at sufficient speed so that it does not accumulate. This situation leads to technologies that do not work with data stored in memory, but with data streams received in real time. This paradigm is becoming more frequent, as the devices, machines, or sensors that generate the data are networked and send their information that must be processed before it is stored.

In 2015 I started working in the area of data analytics. I joined the Center for Open Middleware (COM), a joint research center by the Universidad Politécnica de Madrid and Banco Santander, to make my master thesis. The collaboration between the industry and the university in this research center allowed the creation of interesting projects both for the academic world and the business world.

One of these projects consisted of designing a tool to help the management of distributed systems by predicting failures. The project's objective was to create an Online Failure Prediction (OFP) artifact based on Machine Learning (ML) algorithms. Several of the research lines started at this center have continued active years later. The research work carried out in this thesis is the continuation of the line on predicting failures.

The proactive management of failures saves companies enormous costs [2]. Cloud computing means that this proactivity has been simplified in many scenarios where it is possible to assume the appearance of failures thanks to the redundancy of the systems or the service's replication speed. However, it is not always possible or profitable to have the necessary redundancy or restart business processes. On these occasions, it is still critical to anticipate failures to avoid them or mitigate their effects. Distributed systems such as computer networks are composed of related parts. These relations often imply that when a part of the system fails, the effects are observable in other parts of the system. The observed effects can be interpreted as patterns, and in some cases, the presence of these activity patterns can be used as indicators to predict the occurrence of a failure. Machine learning can create failure prediction models using historical data from previous failures identifying the named patterns. Although this technique is significant, the proper process is not always followed to generate these models, and in some cases, the performed steps are not correctly reported.

In this work, I intend to define a generic methodology to train this type of models, based on the published research related to the prediction of failures. This methodology intends that it can be used as a guide in failure prediction projects, regardless of the application domain. Data analytics projects, and more specifically, machine learning modeling projects, must follow a process of data processing and preparation. This process has a significant impact on the project's outcome, but unfortunately, there is no single universal solution applicable to any environment. The methodology I propose in this research is defined as the series of data processing steps that make up the usual process but focused on failure prediction projects' characteristics. To validate the methodology I propose and the contribution I named later, I use an anonymised dataset provided during the COM project I previously named. This dataset is extracted from a real industrial network.

Prediction models work because they can detect patterns of activity in the target system. In the case of machine learning, these models learn how the system works. The second part of my research is the proposal to use these modules that have learned how the system works to generate new data that emulates the system's

behavior. That is, create a simulator. One of the most common reasons why fault prediction publications do not detail the processes performed or the data used is that this data contains sensitive information to companies and are reluctant to publish it. Another difficulty in reusing published work is that the algorithms, techniques, and tools used are dependent on the work domain. A simulator that generates such datasets can be used to generate data with which the results of different investigations can be compared.

1.1 Methodology

The research, solutions, and conclusions I present in this document summarizes the work I have done during my Ph.D. This document gathers the two main contributions produced during this period of study of the area of failure prediction in computer networks. My contributions are proposals to solve open problems found in the review of area literature and problems encountered when working with a real data set provided by a company. Therefore, I combine an academic approach with an industrial approach in my contributions. The methodology used is also a combination of several known methodologies that I describe in the following sections.

1.1.1 Knowledge Discovery in Databases

Knowledge Database Discovery, or KDD, refers to the overall process of discovering useful knowledge from data. The KDD methodology was proposed in the late 90s [3], [4], then the process has evolved and has been adapted to new research fields as machine learning, data visualization, high performance computing, or scientific discovery. Many other methodologies take KDD as the base to define their process. Some examples of this are the CRISP-DM [5] or the data science methodology [6]. In every case, the core of the methodology is the same. The changes are focused on the final target or in specific steps.

There is no universal definition of the KDD process provided that the domain or the goal affects the definition itself. I will cite and explain the definition proposed by Fayyad et al. in [7] as it is independent of the application domain or the goal. They define the KDD process as *The nontrivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data*. The KDD process is schematized with a block diagram, as shown in Figure 1.1.

Figure 1.1: Knowledge Discovery in Databases process diagram

The process is a set of forwarding steps, with the particularity that loops or reversals can occur between any pair of steps. The leading cause of these loops is that in a process like KDD, the decisions or actions taken should be based on acquired knowledge or the data itself. These decisions make the KDD process interactive and iterative, which steps can be summarized as:

1. Learning the application domain: each domain could be different and should be studied to acquire some prior knowledge and to define the goals of the application
2. Create or gather a dataset: this is the step of data collection and selection. Not all the data may be useful to the application, and in that case, it should be discarded.
3. Data cleaning and preprocessing: data collected from the real world is usually incomplete, with errors and in a format not suitable for data analysis.
4. Data reduction and projection: This data analysis process seeks to find useful features to represent the data. This operation can be done by Feature Selection, Dimensionality Reduction, or Transformation methods.
5. Choosing the function of data mining: this step is to decide which should be the purpose of the model obtained in the data mining process. E.g. summarization, classification, regression, or clustering
6. Choosing the algorithms: this step includes select which algorithms should be used and what parameters they should have.
7. Data mining: includes the search for patterns of interest, usually in creating a model with the chosen algorithm.
8. Interpretation: the output of the models or the discovered patterns must be interpreted. This step usually causes the return to a previous step and creates visualizations of the extracted information or applies some transformation to remove redundant or irrelevant results.

9. Using discovered knowledge: this includes incorporating the acquired knowledge in the process, taking actions based on that knowledge, or report or publish the results.

This definition uses the Data mining as the tool to solve the problem. This technique can be replaced depending on the goal of the application without affecting the process overall. For example, it can be replaced with Machine Learning, and the only change would be on step 7, which becomes: train the machine learning model.

The main advantage of a KDD process is the flexibility that makes it adaptable to the data. When a project is started is impossible to know all the details and possible outcomes that the data analysis will produce. That situation is why this iterative process is so powerful; It can be adapted to the knowledge acquired during the process, enabling going back, changing things using this new information, and advancing again.

1.1.2 Design Science Paradigm

Research in the field of information systems is characterized by two approaches: behavioral science and design science. The first one seeks to create theories to explain or model human or organizational behavior. The second one seeks to create new techniques to improve our organizational capabilities by creating new solutions, named artifacts. My research has been motivated by industrial needs, so my approach to this work is closer to the second one.

The design science in information systems methodology is profusely described in [8]. The main idea of this methodology is the advance of human knowledge by the generation of artifacts. These artifacts can be produced in different structures such as software, algorithms, mathematical procedures, or just new theorems or ideas. This methodology proposes a framework to understand the researcher's place between the industrial and knowledge bases. Figure 1.2 shows a simplified diagram of this framework. Each of the external actors provides something to the researchers and takes something from them. In the case of the industrial world, by studying the business needs, the researchers can define and model existing problems, and their return is the proposals of solutions for these problems in the shape of artifacts. The researchers use the prior knowledge defined in scientific publications to solve these problems, the knowledge base. With this knowledge, the researchers can create reliable and rigorous artifacts and create new knowledge incorporated into the knowledge base. The way to incorporate this new knowledge is by communicating it, usually in scientific papers.

Figure 1.2: Framework for Design Science in information systems

This methodology is designed to be a problem solving process. It should be adapted to many specific situations, so instead of defining a strict set of steps, the methodology is structured as an enumeration of 7 guidelines. These guidelines guide how the problem should be designed based on the acquired knowledge and its application to build and evaluate an artifact. The guidelines cover all the phases of the process to define the artifact solution for a specified domain, the artifact evaluation, knowledge contributions, and research communication.

- **Guideline 1: Design as an Artifact**
This guideline states that the result of a design science research is a proposed artifact created to address an organizational problem. The artifact should be rigorously and precisely defined for the application of the specified domain.
- **Guideline 2: Problem Relevance**
The research objective is to acquire and understand the knowledge that enables the development and implementation of technology-based solutions for unsolved business problems.
- **Guideline 3: Design Evaluation**
The artifact's utility, performance, and quality should be demonstrated with precise evaluation and validation methods. These methods should be defined in a way relevant to the domain and should include integrating the artifact on it.
- **Guideline 4: Research Contributions**
The novelty or innovation of the proposed artifacts should be well defined. One of the research process goals is to create new knowledge, and this methodology is not an exception. The contributions can be the artifacts themselves, the solution's design process, or the evaluation methods.
- **Guideline 5: Research Rigor**
Rigor should guide the decisions taken in the design process and how research

is conducted. A design science research requires the application of rigorous methods in the artifacts' creation and their evaluation.

- **Guideline 6: Design as a Search Process**
Design science is an iterative process. The artifact design should be a process of search and test different solutions available in the knowledge base to find the one that fits best the problem needs.
- **Guideline 7: Communication of Research**
Design science research results should be presented both to the industry as well as to the scientific community. That communication is usually addressed by publishing the results in scientific papers.

1.1.3 Followed Methodology

The methodology I have applied for this doctoral thesis development is a combination of the two methodologies described in this chapter. I have applied the design science methodology for problem identification and objectives and the KDD methodology to design artifacts or solutions.

This work's motivation arises in a project developed at the Center for Open-middleware during the years 2015-2017. This project's objective was to develop a network failure predictor that worked at run time, i.e., an Online Failure Predictor (OFP). To achieve this goal, the bank collaborating with COM ceded monitoring data from one of the production subnets. This data was anonymized to respect the confidentiality of the company.

The objectives were defined according to the project's requirements and a review of the existing literature on the prediction of failures in distributed systems. My master thesis consisted of the development of the first version of the predictor in real time. The architecture of this predictor was later published [9]. During the development of this artifact, when working with the data, new open problems arose. These problems were contrasted with existing literature, and new solutions were proposed, which were also published [10], [11]. The contributions presented in this thesis have been identified in the same way, in an iterative process of working on the solution, acquiring knowledge, and applying that new knowledge to the solution artifact in the next iteration. Both KDD and design science are iterative methodologies allowing them to make data-driven decisions.

The first of the contributions respond to the need to unify all the knowledge acquired in the processing of data for the prediction of failures and the lack of

a general procedure in the existing literature. The work we carried out corresponded to different stages of the process of preprocessing the data and training the prediction models, however, these works were designed separately, and no way was proposed to unite the different results. My first contribution uses all lessons learned from previous research to propose a general methodology for generating failure prediction models. This contribution is described in Chapter 5. The definition of the second contribution also arises from the knowledge acquired during the investigation. The amount of data we had was relatively small, which limited some of the experiments we conducted. Not being able to access the data source in real time also limited the tests we could perform with OFP solutions. Therefore, combining the techniques that existed for synthetic data generation and simulation with the acquired knowledge of fault prediction, I defined the simulator's contribution in Chapter 6.

1.2 Document structure

This thesis document is divided into eight chapters, followed by three appendices. This first chapter aims to introduce the research's main topic and explain the methodology I have followed to reach the contributions. Chapter 2 describes the State of the Art analysis results, identifying the most common problem solving strategies such as addressed in this document. Chapter 3 summarizes the main open problems encountered in analyzing the literature presented in the chapter above. Besides, the thesis's objectives are defined, and the requirements that I impose must meet the proposed solutions. Finally, I include a summary of the infrastructure used to develop and test the contributions' solutions. In Chapter 4, I describe the data scenario used in this work. Chapter 5 describes a proposed methodology for creating machine learning models for failure prediction. Chapter 6 proposes a fault data simulator architecture based on the prediction models obtained in the previous chapter. Chapter 7 deals with the validation of the results obtained in the previous two chapters. Finally, in Chapter 8, I detail my conclusions and the most interesting future lines of research identified during the development of this work.

The appendices accompanying the main chapters contain details of the tools, techniques, and algorithms used in contributions, as well as detailed results. Appendix A lists all the modules, libraries, and frameworks used for solution development. Appendix B focuses on the algorithms, machine learning models, and data processing techniques used. Finally, Appendix C contains the complete results in table format.

Chapter 2

State of the Art

In this chapter, I present the literature search results that I have carried out for my research. The literature examined is mostly scientific articles published in journals through peer review. In the references analyzed, I have also consulted books and tech reports, the latter being the case of some of the surveys cited. The exploration has been carried out with specialized search engines such as IEEEExplore or Google Scholar, and by analyzing the references of the articles studied and those that cited them. The chapter is structured, starting with an introduction that presents the problem of prediction, its relationship with machine learning, and its applications related to preventing failures. A specific section is dedicated to prediction projects in machine learning, their challenges, solutions, and most used techniques. Similarly, another section dedicated to the generation of synthetic data, focusing mainly on simulation techniques. Finally, the last section discusses the concepts seen, the open problems encountered, and the research lines that can be followed.

Machine Learning is a set of techniques based on the idea of creating a model by training it with samples from a dataset so that this model can learn and generate outputs that replicate the observed data's behavior. Kotsiantis published a review [12] proposing a taxonomy for classification algorithms in Machine Learning, defining Machine Learning as *Inductive machine learning is the process of learning a set of rules from instances (examples in a training set), or more generally speaking, creating a classifier that can be used to generalize from new instances*. This observation-based learning is an automatic process, usually based on statistics. The generated models are evaluated based on whether the outputs they produce, follow the expected behavior, with the particularity that the inputs with which the model is fed can be samples that the model did not observe during its training. That is, models should be able to generalize and respond to situations not covered by training data.

Machine Learning goals are very diverse, but prediction problems have been linked to this field since its origins [13]–[15]. The concept of using the learning power of these algorithms to discover hidden relationships or patterns in the data is interesting in many areas.

The concept of prediction does not have a universal or formal definition, but in the vast majority of situations, it is possible to understand the prediction about the announcement or conjecture of a future fact based on the knowledge or experience of the present and the past. One of the future characteristics is uncertainty; therefore, any solution provided to a prediction problem will never be accurate but will be an estimate of the most likely outcome. Another feature is that, in general, short-term predictions are more likely to be met than long-term predictions.

The requirement to create prediction models is to have historical data to analyze and train the model. This data will have a different nature depending on the area to which they belong, and therefore the objective to be predicted will also be diverse. In medicine, the predictions help professionals be diagnostic [16] or even measure the risk of some treatments [17]. Machine Learning prediction in economics has been used to estimate the investment return in advertising or model the dynamics of emergent economic [18]. Several geology examples use prediction to try to forecast earthquakes [19], [20]. In areas closer to information technologies, ML prediction has been used for predicting failures in telecommunication infrastructures [21], predict defects in software [22], network, and memory failures in supercomputers [23], or fraud detection in web payments [24]. Another typical challenge, related in general to the latter field, is failure prediction.

Failure prediction aims to create models that can anticipate a failure in a software or hardware component. Salfner et al. [25], based on an extensive analysis of existing literature, proposes definitions for three essential concepts of clarification in this field.

- Failure: “an event that occurs when the delivered service deviates from correct service.”
- Error: “an error is the part of the total state of the system that may lead to its subsequent service failure.”
- Fault: “faults are the adjudged or hypothesized cause of an error – the root cause of an error.”

Understanding these three concepts is necessary to know the limits of failure prediction problems. These types of models try to detect these deviations from the correct functioning of the system. For this reason, in some cases, these models rely on anomaly detection techniques [26], [27]. These models' goal is to detect when these failures will occur, not to determine what is the cause of these (determine what the fault is) or prevent them. These two branches, along with failure prediction, complete a Proactive Fault Management system:

1. Failure Prediction: This is the main branch of these systems, which, as already named, is the detection of future failures
2. Proactive response to failures: this branch tries to solve the problem of how action should be taken in the event of a failure to avoid its effect or mitigate it; what actions should be taken and how they should be taken
3. Diagnosis or Root Cause Analysis: Using historical logs and records of predictions made, this branch aims to discover what caused the Failure

While closely related, these three problems are very different, and each of them requires a detailed study. In this document, I focus on the first of them.

There are failure prediction works in various domains, such as the prediction of hard drive failures [28], supercomputer failures [29], hardware components failures [30], telecommunication system failures [31], or distributed system failures [32].

Prediction on distributed systems is usually based on the data collected by monitoring a system's network. This monitoring data usually includes error reports, monitoring events, or failure detection events themselves. Predictors feed on this information to find activity patterns that precede failures [33]–[35].

Whatever the domain, all prediction models need a dataset to train. This dataset must represent the usual system activity, but it must also include situations where failures occurred. Depending on the level of system activity this set is working with, this data can span a few days of data up to several years. Raw data typically cannot train a model, so there is always a preprocessing stage and preparation of the dataset [36].

For a system to be functional, the number of failures that occur cannot be vast. Failures are usually rare events, and the dataset that is collected will have many more situations where there are no failures than if they occur.

One resource used to deal with these situations is to generate new data synthetically. These new synthetic samples introduce more fault information to assist

in model training. There are different ways to create synthetic data. The most immediate ones rely on combining existing samples to create new ones. Another more complex option is to use a simulator that replicates the system's behavior or at least the data generation that occurs on it [37]–[39].

Simulators also have varying degrees of complexity. The simplest ones are limited to generating data in the same format as the original data. Others try to replicate the internal relationships between variables. Furthermore, finally, the more complex ones directly simulate the system that generates the data. Simulators are a broad field of research in which this type of synthetic data generation is just a small branch. Another interesting feature of simulators over other more straightforward synthetic data generation techniques is that they can usually be used to generate the data in real time, simulating the output of the system studied.

On the other hand, simple mechanisms can only generate offline data, generating a whole dataset instead of stream data. This feature makes simulators even more interesting in the failure prediction area, as one of the subfields in this area is real time failure prediction. Having a simulator that generates real time data can be useful for testing how trained models respond at run time.

Not all prediction tasks need to run in real time. For example, some predictive maintenance systems run once a day or a week [40]. The situations in which real time prediction is used are those in which we work with systems that generate a high flow of information and on which we can act at run time. That situation can be understood as the models' input instead of being a sample will stream the flow of samples. Likewise, the output of the model will be a data stream with the predictions made. This mode of operation involves performance requirements for models that can be summarized in that a model must be able to generate an output corresponding to an input sample before the next input sample arrives.

2.1 Machine Learning Failure Prediction

Prediction projects in Machine Learning have a feature that sets them apart from the rest. In this type of problem, the temporal dimension of the data should always be taken into account. Predictions are answers about the future considering the present and the past. This time dimension may be lost during data processing, and the input to the models may not have any time mark, even in these cases, time will be contemplated during processing. Historically the usual technique for working with time-dependent data has been time series; unfortunately, these cannot be applied to all kinds of problems [41]. Since the mid-90s and more often

each time, Machine Learning algorithms have been used to solve such problems. According to Google Scholar, searching for "*machine learning prediction*" in different years intervals, in the last decade, around half a million publications were published matching this search, while in the 90s, there were only 20k publications. This evolution is plotted in Figure 2.1. These algorithms generate models using a training process in which a dataset is used. This dataset is typically extracted from the data collected over time and by applying aggregation methods or snapshots to form a static sample set that is not necessarily ordered chronologically. The temporary representation of the data used in each project depends on the objective and application domain. In either case, the process is usually called data windowing.

Figure 2.1: Growth of the number of publications in Machine Learning prediction since 1980

2.1.1 Data windowing

Data windowing refers to the process of capturing the state of a system by observing its state over a defined period of time, called a time window [42]. The time window is characterized by its length. This length is usually a time dimension (e.g., 5 minutes or 1 hour). In some cases, when the data is presented in the form of discrete events, the length or capacity of the window can be measured in the number of samples [43] (e.g., the last ten events occurred in a monitoring system or the last five samples received by a sensor).

The data collected in the time window is aggregated and represented as nu-

meric variables. The aggregation that is performed depends on the problem. In some cases, it will be an event count, a moving average, or other domain-specific aggregation types. In the prediction tasks, the window with which the models' input is built is known as the observation window. For these cases, it is necessary to define a window that specifies the output or prediction's validity time. This second window is called a prediction window or prediction time [25].

Figure 2.2: Online Prediction schema. Figure extracted from [25]

Time relations in online failure prediction. Present time is denoted by t . Failures are predicted with lead-time t_l , which must be greater than minimal warning-time t_w . A prediction is assumed to be valid for some time period, named prediction-period, t_p . In order to perform the prediction, some data up to a time horizon of t_d is used. t_d is called data window size.

The observation window represents the time that the model observes the system to calculate its output. The length of the observation window determines how much information the window can contain. This length depends directly on the nature of the data. In some cases, it may be seconds [44], hours [45] or even days [46]. Calculating the right dimension for each project is an open problem in the literature for which there are various proposed solutions. In the most straightforward cases, the authors decide the window size, and no discussion is presented about its influence [47]. Other publications perform grid searches to study the influence of window length on model performance [48], [49]. Finally, some works apply more advanced techniques to optimize the window size in the problem studied. An example of these works is the research published by Weiss et al. among several papers [33], [50], [51], which proposes the use of a genetic algorithm to determine the best window size based on the data.

The prediction window size also depends on the problem studied and can again range from minutes to days, or more. Predictions of hard drive failures or hardware components may have time prediction windows of several hours [45], [52], while intrusion detection in cloud systems should predict their outcome in minutes [53]. In some cases, such as those related to fault prediction physical components, the window may not have a temporary dimension but may have a length measured in the number of uses [54]. In [55], they propose a different approach is called event driven prediction in contrast to the classical one, which the named window

prediction. This approach consists of eliminating the prediction window making the predictions valid in only one instant of time, separated from the observation window by a time called Lead Time.

2.1.2 Unbalanced data

Regardless of the problem's domain or objective, one of the main problems to deal with in prediction work is the scarcity of samples that are intended to be predicted. Elements that are attempted to predict are often unusual events. Therefore, the data used contains many more negative samples, in which the result to be predicted does not occur, compared to positive samples, in which if the result of interest occurs. In these cases, it is talked about the dataset being unbalanced or simply an unbalanced problem [56].

This lack of balance between positive and negative samples affects many stages of the process of generating a Machine Learning model. Some of the algorithms commonly used for generating prediction models, such as Naive Bayes or Support Vector Machine [57], assume that the data used during training is balanced. When working with an unbalanced dataset, they may fail to learn how to estimate the importance of patterns generated by positive samples or even consider some patterns in the positive class as noise [58]. Thus, to be able to evaluate the performance of the prediction models correctly, it is necessary to apply some process that balances the classes of the data or to use algorithms that support this problem [59]. The second option's biggest drawback is that the range of options is limited, and the number of job references that use them is small, which can lead to a lack of support in the literature for justification.

The most common solution to dealing with an unbalanced data problem is to resample the data [57], [60], [61]. Data resampling involves adding or removing samples from the dataset until there is a balance between classes. In the case of adding samples (of the positive class), it is considered an oversampling process, in the case of removing samples from the negative class, undersampling. There are also hybrid methods that combine the two cases.

2.1.3 Evaluation metrics

The techniques used for evaluating prediction models are diverse. Depending on the problem, you try to solve different metrics that can be used according to the objective sought or different interests. However, when working with unbalanced datasets, these metrics' choice has to be more cautious since the difference between the number of samples in each class can affect the representativeness of the metric [62]. A typical example for illustrating this situation is the accuracy metric in classification models [63]. This metric is calculated as the number of times the

classifier has generated a successful response about the total number of responses. If the classes are balanced in the samples with which the model is evaluated, there is no problem.

Nevertheless, if any of the classes predominate over the rest, it may be the case that the model works very well with the majority class and very severely with the minority class, which is usually the one of interest, and yet the accuracy metric is very high. Therefore, the metric used to evaluate the model should take this effect into account. When working with binary classification, several options are frequently used.

Despite the example described above, the accuracy metric is commonly used due to its easy interpretability [64]. Another simple metric used is precision, which is calculated as the correctly predicted positive samples' ratio over all the positive samples. This metric is proportional to the number of correct positive predictions, so it is useful when the number of false positives is not important. The data imbalance can cause that even with high precision, some rare events remain unpredicted, not considered by the model [65]. Usually, the precision is given with the recall metric to provide a more complete vision of the model performance. The recall is calculated as the ratio of the correct positive predictions over the overall positive predictions. These two metrics together give a reasonable estimation of the model performance, with the drawback of being a tuple two values is more challenging to compare than metrics with one numeric value. The F-Measure, or F-Score, is a metric that solves this drawback. It is calculated as the harmonic mean of the precision and recall. According to [64], *F-Measure is commonly used and is more informative about the effectiveness of a classifier on predicting correctly the cases that matter to the user*. The above metrics can be calculated with counts of the classification results. Metrics such as the Operating Receiving Curve (ROC) are calculated as the curve obtained by modifying the model's decision threshold. This curve's axes are the ratio of correct positive predictions (the recall) and the ratio of the correct negative predictions. To represent the curve as a numeric value, the area covered is calculated as the metric. However, this metric has been proved to be too optimistic in unbalanced datasets [66].

Choosing any of the above metrics should also be noted that prediction tasks are often complex problems and the ranges of metrics are not usually as high as in other areas of Machine Learning. Predicting telecommunication equipment failures in [67], the authors achieve a 3% precision and recall of 63%. [28] achieves a 30% recall with a 0.3% false positive ratio detecting failures in hard drives. An F-score of 48.21% is achieved, predicting failures in data centers in [68].

2.1.4 Algorithms

In choosing the algorithm with which to generate the prediction model, there is also a wide variety of options. This range of algorithms is conditioned on how the model should express the prediction. In some cases, the prediction can estimate the value of a variable, such as the case of the prediction on the stock exchange [69], [70]. However, in some areas such as failure prediction or events in general, the prediction is expressed as a binary classification task. This binary prediction takes only two values, positive and negative, predicted that the event will happen or will not happen. For binary prediction options starting with classic Machine Learning algorithms that attempt to model data mathematically or statistically, either by inference such as Naive Bayes, by looking for a relationship between features and the target variable such as Logistic Regression, or by finding a hyperplane that separates the possible values of the target variable such as Supported Vector Machine or SVM. The family of tree based algorithms is one of the most commonly used options, proven their effectiveness in prediction based on event patterns [71]–[73]. The basic algorithm of this family is the Decision Tree. The rest depends on it either with modifications or by adding several trees in an ensemble such as Random Forest or Gradient Boosted Trees. In Appendix B, I include a detailed description of each of these algorithms, along with other concepts related to training and evaluation of them. Finally, many authors choose to create a modification of these general algorithms, adapting them to the specific situation of the problem being solved [74]–[76].

Not all these algorithms are affected in the same way, but various sources in the literature emphasize the influence of preprocessing training data on model efficiency [77], [78]. One of the points of preprocessing is reducing noise or removing irrelevant information. This process is usually done by reducing the dimensionality of the models' input, which can be based on combining features or projecting them into a new space, or eliminating those that do not provide information, i.e., feature selection. Works that include this step highlight the importance of applying it and its effect on models [79], [80].

2.1.5 Data availability

One of the problems that can be seen in this field is data accessibility. Many published research results use private data. Application scenarios are usually also very specific [81], [82]. These circumstances make it difficult to compare the progress made in the area.

An exception to the rule is the case with the Blue Gene dataset. IBM Blue Gene/L is a supercomputer of Lawrence Livermore National Laboratory (LLNL), consisting of 128K processors. The Reliability, Availability, and Serviceability (RAS)

produced in the supercomputer are collected into a log file through a monitoring system. The log file with these RAS events is published and can be openly accessed, enabling many research works based on this dataset. Some of these works are focused on the prediction of failure events in this scenario [23], [55], [83], [84]. The advantage of having a public domain dataset allows comparing the research made by other investigators. The use of domain-specific features or very precise details of the data can improve results in a particular case, but significantly limits the application of proposed solutions to other domains.

2.2 Simulation

In the previous sections, I have named the need for sufficient data to train efficient models, particularly the problem of the imbalance of data, especially in prediction projects. One of the most commonly used solutions to address this problem is the generation of synthetic data. Synthetic data generation can be carried out in various ways, such as combining existing data to generate new samples, such as SMOTE. SMOTE [85] is an algorithm commonly used in data imbalance problems both to create samples and to delete them, i.e., oversampling and undersampling. The idea of this algorithm is to create synthetic samples by interpolating existing samples in feature space. This technique has been successfully used to improve prediction algorithms' performance, such as in [86] that it is used in the prediction domain of the software compilation result. Another more complex option is to try to replicate the operation of the system studied by simulation. Simulation tools apply to a multitude of fields. The goal is not always to generate synthetic data, but to be used to test the operation of models or artifacts. This literature review will focus on the uses of simulation for data generation and more in detail data for training and evaluating machine learning models.

These simulations can try to emulate the complete system's operation or focus on the injection of the data to be modeled. In [87], a benchmark of different machine learning algorithms is carried out to predict software failures based on the amount of synthetic data inserted. To control the amount of synthetic data, they use a tool [88] that injects faults in the system's regular operation. Other options emulate the system's behavior, highlighting the simulators whose objective is the generation of events of interest to the problem. This area is known as the Discrete Event Simulation [89], [90]. This type of simulation attempts to replicate the conditions that lead to the generation of events. This technique goes back several decades, with works in the 70s proposing efficient data structures for dynamic event generation [91]. The main disadvantage of this approach, as some authors point out [92], it is necessary to have an exact knowledge of the system

to be modeled. This is an option when the work environments are simple or well bounded, but real systems, such as distributed systems, are very complex and are not easy to model mathematically. In these cases, agent-based simulation stands out as a good alternative [92].

Agent-based simulation appears in the literature under different acronyms: Agent Based Modelling (ABM), Agent Based Simulation (ABS), Multi Agent Based Simulation (MABS) or Agent Based Modelling and Simulation (ABMS). This simulation is based on the concept of software agents proposed in the 90s [93], [94]. The foundation of these systems is the combination of agents that solve simple tasks to build a complex system capable of making advanced decisions. In the simulation, agents are used in the same way. Each agent is responsible for simulating a small part of the simulated environment, and by communicating between these agents, the simulation of the entire environment is achieved [95]. Agent-based simulation has been applied to different areas as reflected in the survey [96]: Biology, consumer markets, supply networks, human movement patterns, high performance computing systems (e.g., Blue Gene), etc. [97] proposes the generation of fraud data and mobile payments with an agent-based simulation in which agents model the behaviors of fraudsters.

Literature combining agent-based simulation (and simulation in general) with machine learning techniques is an area that had not been explored until a few years ago, as some authors stand out [98]. In their article, the authors of [98], propose using agents that can learn from the data obtained from the real world to model the agents' behavior in data simulation. More recent works like [99], propose using agent-based simulation for the generation of training sets for neural networks. In [100], decision trees are used in the agents to carry out simulations on expanding the cholera contagions. [101] proposes using algorithms that allow agents to perform reinforced learning based on the data they observe during the simulation.

Data simulation applications for machine learning are not limited to, improving model performance. They can also be used to test the systems that implement these models. [99] proposes the use of a data generator for benchmark applications in the realm of big data. Some authors [102] also advocate using synthetic data in training both model validation and synthetic data. It is possible to ensure that situations present certain conditions, unlike the actual data on which there is no possibility of this control. Some works propose the use of machine learning algorithms to explain the complex behaviors emerging from agent-based simulations [103]. Other works, such as [104], use simulations in combination with machine learning to perform a risk assessment.

2.3 Discussion

Machine learning prediction tasks apply to a variety of fields, including failure prediction. The amount of published literature confirms the validity of this approach. However, the process, even starting from the stages typical to any machine learning project, the complete process is not usually described with the same detail as other machine learning projects. In the bibliographic review, I have done, I have not found any work that addresses the whole process of generating the prediction model since the raw data acquisition. Many examples concretely detail different stages of this process, such as the algorithm benchmark, data imbalance problem, or the evaluation metric's choice. Even some publications do not specify anything about the data preparation, although it is proved that this process affects the quality of the results obtained when modeling them [105]. To illustrate this, in the following table, Table 2.1 on page 20, I created a comparison of a sample of failure prediction publications. In this comparison, I analyze which stages of the process of creating a prediction system are covered in the articles and which are not named or not detailed. In no case is it my intention to say with this comparison that the articles included in it are incomplete or not valid. All of them are contributions to knowledge published in scientific journals with their corresponding validation methods. I just want to highlight the fact that publications in this area generally focus on specific aspects of the process.

ref	P	DR	AB	MD	U	Topic
[31]	+	-	-	+	-	Telecommunication Equipment Failures
[48]	~	+	-	+	-	Computer System Failures
[106]	~	-	~	-	-	Identify and recognize patterns of errors that lead to failures
[83]	+	+	+	+	-	Failure prediction in BlueGene/L
[32]	-	-	~	-	-	Failure prediction in online-services during their real-time lifecycle
[40]	-	+	+	~	~	Predict rare failure events in manufacture
[35]	~	-	-	+	+	Failure prediction in distributed business processes
[24]	+	+	+	-	+	Fraud Detection Applied to Web Payment Systems
[55]	+	~	-	+	-	Comparison of the event based prediction with observation window prediction
[107]	-	+	-	+	-	Online failure prediction in cloud datacenters
[108]	~	~	-	+	-	generic algorithm of online failure prediction
[109]	~	+	+	+	~	Switch Failure Prediction

+: named and detailed, ~: named but not solved or described, -: not named

P: Preprocessing, **DR**: Dimensionality Reduction, **AB**: Algorithm Benchmark, **MD**: Metric Discussion, **U**: Unbalance

Table 2.1: Comparing addressed points in published works on failure prediction

This article [31] proposes using a genetic algorithm to determine the search space for event patterns that cause telecommunication systems failures. The article proposes the use of pattern mining; therefore, it is not applicable to address the problem of imbalance in this project. The article also includes an interesting

discussion about which metrics are useful for evaluating prediction models. [48] proposes a predictor of failures in computer systems with SVM based on event patterns. There is a grid search process to find the best observation window, but this grid's intervals and limits are not argued and are taken based on the authors' opinion. [106] is a proposal to use Hidden Markov Models (HMM) in fault prediction problems. This proposal argues the viability of HMM in such problems but does not compare it to other options. It also names the preprocessing to be performed on the training data but does not detail it. [83] is about predicting failures in Blue-Gen/L. The authors propose an aggregation function to create the model input, but the function is problem-specific and is justified on domain knowledge. The paper includes an excellent algorithm benchmark between RIPPER, SVM, KNN, and their customized KNN, with a good discussion about which metric they used to evaluate the models. The authors of [32] propose the use of synthetic data to train a predictor model based on a custom classification algorithm. Although it is compared with other algorithms, the comparison is not rigorous as they used the accuracy to evaluate the model, and some of the algorithms compared are not detailed (e.g., Neural Networks) or have different goals (e.g., Clustering). [40] uses the feature importance to perform a feature dimensionality reduction, and the authors give an argument and detailed algorithm comparison. However, the imbalance problem is named but not addressed. [35] is a proposal for failure prediction in distributed business processes in which the authors use a controlled failure injection to deal with the data unbalance. They also discuss the correct metric to evaluate the models among the precision and recall or the MCC [110]. [24] is work on Fraud Detection Applied to Web Payment Systems, where feature selection and undersampling are used to improve the models. [55] is a great example of a study of the observation window effect in predictions, but it is only studied with one algorithm. In [108], the preprocessing and the dimensionality reduction are named but not detailed. [109] proposes a feature aggregation function to reduce the number of input variables and deal with the unbalance. However, this aggregation and many decisions taken are based on domain knowledge and not justified, making them difficult to reuse.

The most common metric in these articles is the F-Score, which argued that it is a balance between accuracy and recall. Only a few of them include an algorithm benchmark. The unbalance problem is usually named but not always addressed. The feature selection and preprocessing usually are applied based on domain knowledge.

Choosing the algorithm to generate the prediction model is an open problem, and although some results propose modified algorithms, none of them have signifi-

cant acceptance. Therefore, it is still necessary to make a comparison between the most common algorithms for each particular scenario.

Other problems with widely accepted solutions, such as using sampling methods to work with unbalanced datasets or feature selection, have been successfully tested isolatedly. Although many of these new techniques propose interesting advances, the lack of examples of their use in various scenarios limits their possible use. However, they rarely are present together in the same work, so it is difficult to know how they affect each other.

The problem that I previously named in this chapter on the availability of data, especially in problem predicting computer network failures, makes it difficult to reuse published results. The only accessible documentation is the results that are published in the papers, which is usually reduced to the final results without including the middle result. Another fact that is not usually included in the publications are the execution performance of the project. However, it is usual to find the training or prediction times, but as I emphasize, not the entire process.

Concerning simulation techniques, the area of knowledge is pervasive and can quickly move away from prediction problems since its applications are broad. There are solutions for the simulation of data sets in log format, or discrete events in general, the simulation of failure events based on the interaction of events is a little explored field. Some simulation models, such as the agent-based one, incorporate the interaction and sharing of information between different actors.

2.4 Chapter summary

This chapter discussed the existing literature on machine learning prediction and simulation for synthetic data generation. The first section of the chapter summarizes the overview of prediction projects.

In the second section, I discuss the main challenges of prediction projects. The first is the temporary representation of the data by the process of windowing. Continuing the problem of unbalance of prediction datasets and the usual solutions to deal with the problem. Next, I describe the most commonly used metrics to evaluate the performance of the prediction models. I also analyze algorithms and families of algorithms usually used in prediction. Finally, I present the different prediction scenarios, and in more detail, those that enable real time prediction.

In the next section, I present an analysis of the different categories of simulators that exist for the generation of discrete event data. I delve into the use of agent systems to perform these simulations and their advantages over the rest of

the options.

Finally, in the discussion section, I analyze the main challenges and open problems encountered in literature.

Chapter 3

Objectives

In this chapter, I summarize the objectives I plan to solve with the work presented in this document. The narrowing of these objectives is based on the facts found in the literature review detailed in the chapter above. In addition to research objectives, I establish design requirements that force the solutions I propose to be carried out and not just theoretical solutions. Finally, I briefly describe the processing environment I use to process the experiments.

3.1 Objectives

As I mentioned in Chapter 1, one of the motivations of this work was the participation in a research project to create an Online Failure Predictor for a distributed system network. The work carried out in this project and the results obtained later present contributions in different stages of the forecasting model generation project. However, these contributions were designed separately, and the integration of all of them in a single process is a problem that remained open. Results found in the related literature present this same characteristic. The contributions focus on a point in the process, but its integration with the other stages is not detailed.

In the previous chapter, I also mentioned the lack of public data sets, with the notable exception of Blue Gene. Without the original data and only having the final results published in the papers, it is not easy to verify the proposed solutions' validity in specific domains or scenarios. Added to this problem is that datasets, including private datasets, are usually sample collections for a limited period of time, and the original source of the data cannot be accessed. This makes it challenging to test models or prediction systems in real time.

The objectives I set for this work are drawn directly from the problems observed when analyzing the related literature or working within an industrial research project. Being real problems found or raised in the literature, this thesis is not based on a series of hypotheses that validate but on a series of solution proposals accompanied by their validation. With all these considerations, I define two main objectives for my thesis research.

3.1.1 Objective 1: propose a general, documented, and complete methodology to create Machine Learning prediction models for failure prediction

Create a methodology for generating prediction models spanning from raw data acquisition to model validation. The methodology will include the necessary steps for preprocessing the data and training machine learning models. Each step of the methodology will be defined as a modular piece, configurable, and detailed to enable reuse.

3.1.2 Objective 2: design the architecture of a discrete failure simulator based on prediction models

Design a simulator by leveraging the ability to recognize the event interaction patterns of failure prediction models. The architecture will be generically designed but tested with a real industrial scenario.

3.2 Requirements

The requirements below are intended to make others can use the work presented in this document. The proposed solutions will be well detailed, and I will try not to have any dark areas in the processes described. One of the purposes of this work is to be useful and replicable by other researchers, therefore reasonable to impose strict requirements on the development and presentation of solutions.

Automation

The processes that I define must be able to execute with the minimum possible human interaction. Where this interaction is indispensable, advanced knowledge of the techniques used to be able to use the solutions should not be needed.

Scalability

Although the scenario from which the data I use for this job is extracted for this job is a real company network, not all networks have the same dimensions. Solutions must be able to be applied to more extensive networks, higher data volume, or a greater variety of events. The easiest way to ensure this requirement is for solutions to work in a distributed environment. Specifically, solutions must be compatible with Apache Spark, which is currently the most widely used distributed data processing framework.

Understandability

Although the above requirement attempts to minimize human interaction with solutions, it is sometimes necessary or useful. Prediction models can be used to find the causes of some failures, or the simulator can be modified to incorporate new behaviors. In both cases, both models and code must be possible understood. Any part of the solutions must be able to be seen as a white box.

Reliability

The proposed solutions should be validated and tested to provide expected levels of efficiency.

Real data

The solutions must be tested with real data extracted from an industrial environment.

Upgradeable

Solutions must be able to adapt to changes in the scenario without having to be modified.

Expandable

Solutions should be designed modularly to allow them to be expanded with the addition of new algorithms, data sources, or other modifications.

Open Software

The solutions I propose will use only open software. Also, the resulting or used artifacts, code, and data will be published openly under a GLP license.

Compatibility

Proposed solutions should not be dependent on the programming language or tools used. Processes should be able to be implemented using tools similar to those used in the proposed solution.

Agnostic to domain

The data from my scenario was extracted from an internal network of a bank. This data may differ from similar environments: different topologies, different events, etc. Indeed, many of the analyses I perform can get better results if domain knowledge is used. However, by maintaining solutions beyond the actual meaning of events or domain information, solutions can be applied in scenarios other than mine.

Reproducible

All processes and tools (hardware and software) will be detailed to ensure that experiments can be reproduced.

Achievable

This requirement has to do with the necessary resources to carry out the experiments presented. These resources can be both hardware or software, as temporary limitations. Many processes in the area of data processing and machine learning can be enormously expensive computationally. The proposed solutions should consider the balance between the quality or accuracy of the outcome and the cost of obtaining it. The decisions taken in consideration of this balance will be described in detail for future replicas of the solutions that can be adjusted based on the available resources or the desired accuracy levels.

3.3 Processing environment

To end this chapter, I will talk about the environment in which I run all the experiments. The machine specifications do not have any effect on the results, but they affect the computation times. I use two different settings: the first one is a cluster of dedicated servers hosted on my university data center, and the second is a laptop. The cluster is a remote environment with great processing power in which I can run distributed algorithms and explore their horizontal scalability. The local environment is less powerful, but many of the experiments do not require much

computation, and the access to the results is more accessible -from the human point of view- than in the distributed setting.

3.3.1 Environment A: cluster machine

This is the distributed environment or cluster environment. The cluster is made of 8 equal machines with these characteristics:

- *Server Model*: HP ProLiant SL250s. Gen8
- *Operating System*: Arch Linux
- *Processor*: two Intel Xeon e52630v2 2.6GHz, 6 cores each. 24 virtual cores per machine
- *RAM*: 32GB

Most of the experiments run on this environment use Apache Spark which is configured with one machine as master and the other 7 as slaves. Each slave hosts 6 workers with 4 cores and 5GB RAM, making a cluster of 42 workers with 168 cores and 210GB RAM.

Beside of Spark, this environment has the following services:

- *HDFS*: distributed file system with 10.6TB of storage. The original dataset and many of its transformations, such as the different windowing schemes, are stored on this system.
- *MIFlow*: results repository. I store all the experiments reports and results in this system.
- *Zeppelin*: code notebook able to send jobs to the Spark cluster.

3.3.2 Environment B: laptop

This environment is my work computer and in this document, I will usually refer to it as the local environment. The laptop has these characteristics:

- *Model*: Dell XPS 15 2in1
- *Operating System*: Windows 10
- *Processor*: Intel Core i7-8705G 3.1GHz with 4 cores
- *RAM*: 16GB

This laptop is where I have performed most of the experiments that were not related to training ML models, which is probably the most expensive task and that is most often repeated in this research. For performing data scans, result collection, or charts generation this environment is sufficient, but I also got to use it when I had to do some tests that included training models. In these situations, I have found that the run time is around 30-40% slower than running the same process on a single of the machines that make up the cluster. Therefore, a crude approach, only interesting for making time estimations, is that the compute capacity of the cluster is roughly 12 times that of the laptop.

3.4 Chapter summary

In this chapter, I have presented the objectives set for the research work of this thesis. These objectives are based on the revision of the literature studied in Chapter 2 and previous work in the area, as named in Chapter 1. The objectives are supported by several requirements that must be met by the proposed solutions.

Finally, the two processing environments I have worked with to conduct this research are briefly described. The first is a cluster of 8 machines in which I can run distributed calculations to take full advantage of their power. The second is a working laptop in which the simplest operations are performed

Chapter 4

Scenario description

All the experiments and validations are based on a dataset extracted from a real production network provided by a Spanish bank. The fact that the data comes from a real infrastructure causes that the raw will not be clean. It needs preprocessing and filtering as not all the information can be used in data analysis, for example, to errors in the measurement quantity of records.

In this chapter, I present in detail the dataset, a report of the analytic exploration which describes its main characteristics and, the necessary cleaning and transformation I made to obtain a usable format for the experiments.

4.1 Background

As I mentioned, the dataset is extracted from a Spanish bank infrastructure. The data was collected by the monitoring software Spectrum CA which generates a log of all Trap messages sent using the SNMP protocol. These messages contain information about alerts, signals, or failures that happen in the network, most of them are from the network layer, but there is also some related to the application layer (in general services or applications that stop working because they lost network access in some way). For confidentiality concerns, all the data fields have been anonymized: all the IPs, machine names, URLs, etc. have been replaced with generic values.

The monitored network is a relatively small subnetwork made of 21 devices. The dataset contains all the TRAP messages produced by any of those devices in the time lapse between the 16th of September, 2014 to the 9th of April, 2015, making a total of 206 days. The messages are collected into monthly log files with a standard format of comma separated values. Joining the eight months logs, the

result log contains 21442 records, an average volume of 104 messages per day. This can be considered a small flow, but it is in the range of what can be expected of a failure dataset, provided that failures should be something sporadic.

All the logs have a common format with ten variables, from those only six of them contains useful information for this work:

1. **Date:** The date when the message was generated. The format is "YYYY-MM-DD HH:mm:ss", e.g., "2014-09-16 04:00:22"
2. **Machine:** The device which generates the message. e.g., "VirtualMachine_01"
3. **Code:** Failure's error code. The format is "0x" followed by 6 hexadecimal values, e.g., "0x21001a"
4. **Message:** Description of the failure. The format is a string of variable length, e.g., "An ospfNbrStateChange trap has been received for router 0.0.0.7. The neighbor router is 0.0.0.3, and the new state is DOWN."
5. **Type:** type of machine which generates the message. It has 3 possible values: router, switch, or virtual machine.
6. **Severity:** severity level associated with the failure code. There are 4 possible values: "Critical", "Major", "Minor", or empty string (which I will refer as "Blank")

Each record in the dataset contains information with the failure code, the machine that generates the message, and the time when it is produced. These three variables answer to what, where, and when, and are the most critical variables in the work presented in this document.

4.2 Event interpretation

To work with this kind of data is as important to know what happens to know where it happens so I created a new variable with the tuple of the Code and Machine values. I named this variable Event, and from this point, I will refer to the dataset as a collection of events associated with the time when they were generated. There are 21 machines, and the distribution of records generated by them can be observed in Figure 4.1.

Figure 4.1: Records distribution by machine

The most active machine is the router; 30% of the records are generated by it. It is also easy to observe that some of the virtual machines almost do not generate records compared with other machines, but, grouping the number of records by the device type as in Figure 4.2, the virtual machines are the type of device that more records produce.

Figure 4.2: Records distribution by machine type

Focusing now on the message codes, there are 87 different values, and their distribution can be observed in Figure 4.3.

Figure 4.3: Records distribution by message code

Finally, the new variable Event has 548 values. To ease most of the processes, I create an index of those values according to their frequency (number of records), making the event 0 the event with the most records and the event 547 the one with the least. Figure 4.3 shows the distribution of event frequencies. In this case, the plot is a line because there are too many events to plot them in a bar chart, but it helps to show that exists a power-law relationship: few events accumulate most of the records.

Figure 4.4: Records distribution by Event

4.3 Dataset description

Known the dataset variables, it is necessary to do a preliminary exploration of the data that will allow us to know different aspects of it and know what kind of information I am going to work with. The first aspect of analyzing is the distribution of events over the monitoring period. Figure 4.5 shows the number of daily events on the network. The distribution consists of a regular component that is overlapped by a series of peaks, highlighting the one that occurs on December 31st. The median number of daily events is 74 and the average 108, while December 24th is triggered to 2618 events, two orders of magnitude above.

Figure 4.5: Daily event distribution over the monitoring period

To observe the usual behavior, Figure 4.6 shows the same distribution but with the peak days filtered. The series has a weekly seasonal component, seeing that the number of events is greater than the working days on Saturdays. This can be seen in more detail in Figure 4.7, a boxplot that shows the distribution of events according to the day of the week.

Figure 4.6: Filtered event daily aggregation distribution

Figure 4.7: Boxplot of event distribution by weekday

Analyzing the time data in detail, this information has a granularity or precision of 1 second. Of the 21442 data records, there are only 10661 different time instants. On average, 2 events per instant of time occur, but the median is 1, the maximum 76, and only in 1170 times more than two events happen in the same second. Given this granularity, two events that occur in the same second are considered to occur simultaneously. The case of the same event occurring more than once in the same second is a limit situation that poses various problems and does not provide relevant information. Therefore, these cases must be filtered so that each type of event is counted once in each instant of time. Thus, of the 21442 records contained in the initial dataset, only 15136 are considered valid.

Applying this correction, the daily distribution of events is represented in Figure 4.8. The first thing that stands out is that the peaks seen before have been reduced by 25% while the rest of the series remains similar to what it was before the correction. So this event duplication may be just failures in the network monitoring due to an overload.

Figure 4.8: Filtered event daily distribution over the monitoring period

Events were indexed by frequency before applying this filtering, and it is interesting to check how many samples each of them loses. Figure 4.9 shows events with at least 10 samples and represents frequencies before and after correction. It can be observed that some events have lost a much higher sample ratio than others. Some like the indexed as 6, 7, 11 or 12, drop from more than 400 samples to less than 100. This means that for those events, the 75% of their samples are duplicates. For other events, this filter does discard only a few samples or even zero. There is no evident relation between which events have the most duplicates and any of their features.

Figure 4.9: Event frequencies before and after the filter

4.4 Data cleaning and preprocessing

One of the critical points of this work is creating models capable of predicting the occurrence of events or generating sequences of pseudorandom events with the same statistical characteristics as those observed in the original data. To successfully create these models, the events I will handle need to have a minimum number of samples because if this number is minimal, the model they generate will be unreliable and also cannot be validated. For prediction models, validation is performed using the K-Folds technique with five partitions. This means that I divide the dataset into five sections. I have to make sure that each of these sections contains some samples of the event I am trying to model. With the purpose of discarding the minimum number of events, I set the threshold to 10 samples. That is, I only try to model those events that have at least ten samples. These thresholds apply to the already filtered data, as explained in the previous section. Of the 548 events, only 168 events have at least ten samples. Figure 4.10 shows the histogram of the number of samples of these selected events. As also seen in previous figures, there are more events the lower the frequency. This implies that many events that have about ten samples are selected and that in fairly likelihood, as has already been named, you cannot generate models that can correctly predict these events. Although I only try to model these 168 events, the data for the unselected events are not discarded, and they can be used to train the models of the selected events. A summary of the selected events relative to the total is shown in the appendix Table C.1.

Figure 4.10: Histogram of event's frequencies

I have calculated how many events occur simultaneously and the average volume of events, but another interesting metric can be easily obtained: the time difference between events. I will go deeper into this concept in Chapter 6, for an exploratory analysis is enough to calculate what is the time difference existing between consecutive events (no matter the event). The median difference is 30 seconds, the average 1561 seconds (26 minutes), and the percent 95% is 7542 seconds (2 hours). These numbers suggest that when the network lowers its activity, the sequences of the events stop for at least some hours.

With this idea, I define the concept of Spontaneous Event: an event which is preceded by no events in the previous four hours. This is an event generated in a steady network, so, in terms of predicting it, I will not be able to find a pattern that precedes it, and thus it will be unpredictable. Applying this concept to the selected events, I get the results to show in Table 4.1. The table shows different counts for each event with at least 50 samples. The full version of this table is included in Appendix C:

- event: event index
- Freq: the raw number of records of the event
- TFreq: the number of samples of an event after the discard of simultaneous ones
- Pred: the number of predictable events, this is calculated as $\text{TrueFreq} - \text{Spontaneous}$

- Spon: the number of spontaneous events and its percentage over the True-Freq

It is imperative to consider the number of predictable samples of an event because it is the maximum number of samples that a model will be able to predict. Consequently, if an event has a low number of predictable samples, its model will be a bad one.

event	Freq	TFreq	Pred	Spon	event	Freq	TFreq	Pred	Spon
0	1208	1194	1097	97 (8%)	33	214	214	213	1 (0%)
1	656	650	634	16 (2%)	34	214	214	214	0 (0%)
2	633	632	523	109 (17%)	35	211	210	195	15 (7%)
3	608	460	455	5 (1%)	36	211	209	209	0 (0%)
4	605	605	483	122 (20%)	37	177	177	161	16 (9%)
5	602	466	464	2 (0%)	38	155	66	65	1 (1%)
8	440	144	140	4 (2%)	39	151	55	55	0 (0%)
9	438	144	142	2 (1%)	41	139	66	65	1 (1%)
10	430	131	123	8 (6%)	42	137	58	58	0 (0%)
13	425	124	123	1 (0%)	43	136	66	65	1 (1%)
14	367	367	299	68 (18%)	46	130	56	55	1 (1%)
15	352	352	350	2 (0%)	48	123	123	123	0 (0%)
16	313	312	310	2 (0%)	49	122	122	113	9 (7%)
17	298	298	297	1 (0%)	50	122	120	120	0 (0%)
18	289	289	280	9 (3%)	51	121	120	120	0 (0%)
19	289	88	85	3 (3%)	52	120	120	120	0 (0%)
20	288	86	86	0 (0%)	53	117	117	116	1 (0%)
21	279	76	76	0 (0%)	54	116	116	115	1 (0%)
22	279	279	177	102 (36%)	55	116	83	83	0 (0%)
23	278	82	82	0 (0%)	56	110	110	109	1 (0%)
24	272	249	249	0 (0%)	57	109	93	92	1 (1%)
25	272	248	230	18 (7%)	58	106	64	64	0 (0%)
26	272	248	230	18 (7%)	61	95	95	95	0 (0%)
27	272	249	249	0 (0%)	69	75	57	57	0 (0%)
28	266	266	263	3 (1%)	75	56	56	56	0 (0%)
29	265	265	263	2 (0%)	76	54	54	53	1 (1%)
30	241	241	240	1 (0%)	79	50	50	49	1 (2%)
31	239	239	239	0 (0%)	80	50	50	49	1 (2%)
32	221	220	220	0 (0%)					

Table 4.1: Event samples summary

In the case of the event with the most repetitions, indexing with 0, the number of positive samples is 1194, about 10% of the total samples. This percentage drops to 0.1% of positive samples in the least frequent events compared to the total. When working with datasets with these characteristics (one class is dominant over the rest), the dataset is said to be unbalanced. This is a significant limitation that

should be considered when applying any process to the results' data or measurement. In particular, this scenario is very unbalanced, and that will carry some problems, for example, unstable model evaluations provided that when a model has only a few samples to check if its output is correct, one correct output up or down has a significant impact in the evaluation.

4.5 Chapter summary

In this chapter, I have presented the scenario on which the research in this document is carried out. The data comes from a network monitoring system applied to a Spanish bank's real subnetwork and production environment. This subnetwork consists of 22 devices, and the available dataset covers a period of 7 months. During these seven months, 21442 alert messages are logged.

Alert messages contain the alert code fields and the machine that generates the message. Joining those two fields brings up a new variable that I call an event. This variable contains information about "what happened" and "where it happened." There is a total of 548 different events. After analyzing and filtering the data, I decided that it will work only with those events that have occurred at least ten times during the observation period. This filtering reduces the number of events to be studied to 168.

Chapter 5

Generic methodology to generate ML prediction models from discrete events dataset

This chapter expands on my first contribution, a methodology for creating prediction models using machine learning that covers all stages of a modeling project. This methodology is defined as a generic set of sequential steps of data preprocessing. I generate incrementally better prediction models with each of these steps, ending with the best models I can create in my scenario. The chapter starts with some definitions and specifications about how the models will be and how they will be evaluated. Then, I describe the methodology structure and present the results of each of the steps applied to the scenario analyzed in Chapter 4. Finally, the chapter ends with a discussion section to analyze the obtained results and the methodology structure.

5.1 Prediction model definition

A prediction model is not a standard concept, and many techniques may be qualified as it. In my particular scenario, I define a prediction model as a classification model that, by processing a numerical input based on the system's past behavior, generates a binary output understood as the occurrence or not of a certain event in the future. Any ML algorithm could generate these models (e.g., Logistic Regression, Random Forest, Support Vector Machine, etc.) as long it creates a binary classification model. The named input will be a vector of non-negative integer values, counting the number of times a specific event happened in a fixed period of time. This process will be addressed in detail in the next section. So, formally I define a prediction model as a piece of software which performs the function f

defined as:

$$\begin{aligned}
 f_e(\vec{x}) &\rightarrow y_e \\
 \vec{x} &= [x_0, x_1, \dots, x_n] \\
 \forall i, x_i &\geq 0 \\
 y_e &\in \{0, 1\}
 \end{aligned} \tag{5.1}$$

The core idea is that the event occurrences in the monitored system are interrelated, and some typical or common patterns could be observed, and by identifying those patterns, in some cases, it is possible to anticipate what will happen next. The reason to use ML is that those patterns are challenging to obtain, and they are not strict in the way that they may have some variations as temporal deviations. Also, even in a not so big system as this scenario, the number of different events is enormous and, try to obtain these patterns, or just guess which events influence the occurrence of other events, is an unreachable task for a human.

Something important to clarify is that I do not try to find a causality relation between events. The patterns I look for with these models identify sets of events that usually preceded other events, not sets of events that cause other events. This may be a faint difference, but some domain knowledge is needed to truly identify the cause of an event. Otherwise, only suppositions can be done. Furthermore, as I stated in previous chapters, all the work done considers the limitation of does not have access to a domain expert to consult.

5.2 Data windowing techniques

The prediction model input design is based on the schema defined in [25] for Online Failure Prediction. The main advantage of using this schema is that they proposed terminology for common concepts widely used in the literature.

Figure 5.1: Observation-Prediction window schema

Figure 5.1 represents an abstraction from the temporal point of view, of the system monitoring. For a given instant t , the next time spans are defined:

- Observation Window: the period of time previous to t that is considered to create the input for the prediction model. This window covers the time range from $t - \Delta t_o$ to t .
- Prediction Window: the period of time subsequent to t in which the output of the prediction model calculated from the input based on the current Observation Window, is valid. This window covers the time range from $t + \Delta t_l$ to $t + \Delta t_l + \Delta t_p$.
- Lead time: the temporal separation between the Observation Window and the Prediction Window.
- Warning time: the minimum threshold value of the Lead Time. This time is related to the necessary time to respond to a prediction. It is useless to know that something will happen if there is no time to react before it happens. This threshold should be defined by a system supervisor, which is a limitation in my work. Based on the events time distribution, I set this value to one minute.

Summarizing, at a given instant t , the prediction model will receive an input with information about what happened on the Observation Window, and it will generate a prediction output which valid since a Lead time lapse during a period equal to the Prediction Window.

With the concept of Observation Window and an indexed list of possible events, the creation of the input, at a given instant, is simple. First, I count how many times each event occurs during the Observation Window. Then I build a vector which, in the first position it will contain the count for the first event in the indexed list; in the second position, it will contain the count for the second event in the indexed list, and so on with all the events. I end with a vector $\vec{v}$ of the same size as the indexed events list, in which the value v_i at the position i is the count of how many times the i -th event of the list occurred during the Observation Window.

5.3 Methods to evaluate prediction models

I use binary classification models as the prediction models, and this has some advantages like the possibility to use the standard evaluation techniques for classification models. In the case of binary classification, most of those techniques are obtained from the confusion matrix:

		Actual value	
		Positive	Negative
Model output	Positive	TP	FP
	Negative	FN	TN

Table 5.1: Confusion matrix

This table counts how many times the model produces a correct or incorrect prediction. For each evaluated sample, one of the 4 cells will be incremented by one: if the prediction (the model output) is positive, a cell in the first column will be updated, or in the second column if it is negative. In the same way, the row will be selected depending on the actual label of the sample. The resulting matrix has 4 values:

- **True Positive [TP]** The number of positive samples which the model classifies as positive
- **True Negative [TN]** The number of negative samples which the model classifies as negative
- **False Positive [FP]** The number of negative samples which the model classifies as positive
- **False Negative [FN]** The number of positive samples which the model classifies as negative

With those definitions, the total number of correctly classified samples is $TP + TN$, analogously the number of incorrectly classified samples is $FP + FN$, and the total number of samples is $N = TP + TN + FP + FN$. In my particular scenario, as is explained in Chapter 4, the dataset is very imbalanced; this causes that TP count is expected to be much bigger than the other counts. Also, the TP samples are the less important situations as they can be understood as "nothing happens, and the prediction model does not generate an alert". In consequence, I should use a metric in which that value has no weight or a small one. Some of the most common metrics obtained from that matrix are:

- **Accuracy:** this is the most intuitive metric. It is a ratio between the correctly classified samples over the total number of samples: $accuracy = \frac{TP+TN}{TP+TN+FP+FN}$. But as I mentioned, the TN value is dominant, so this metric will generate good values even for bad prediction models.

- Precision: this is the ratio of positives samples correctly classified: $precision = \frac{TP}{TP+FP}$. From the prediction point of view, this can be understood as *given the number times the model predicts the occurrence of the event, how many times the event actually happened*.
- Recall: this is the ratio of positives samples correctly classified: $recall = \frac{TP}{TP+FN}$. From the prediction point of view, this can be understood as *given the number of occurrences of an event, how many times the model predicts the occurrence of the event*.
- Fall-out or False Positive Rate: it is calculated as $FPR = \frac{FP}{FP+TN}$. This ratio indicates the number of false alarms generated by the model.
- F-score (or F1 score): this metric is calculated as the harmonic mean of precision and recall $F - score = F1 = 2 \frac{precision * recall}{precision + recall} = \frac{2 * TP}{2 * TP + FP + FN}$. The F-score is a very convenient metric for my scenario, provided that it takes into account all the relevant values from the confusion matrix. **This is the metric that I use to score the models.**
- Cohen's kappa coefficient: this is the most complex metric in this list. Nevertheless, it's equation is also a fraction. It can be interpreted as how good is a model compared to a random model. The formula is $\kappa \equiv \frac{p_o - p_e}{1 - p_e} = 1 - \frac{1 - p_o}{1 - p_e}$, which depends on two values p_o and p_e . p_o is equal to the accuracy which can be interpreted as the probability to give the correct answer. p_e is the probability to give the correct answer by random and it is calculated as $p_e = p_{YES} + p_{NO}$ (or $p_{POSITIVE} + p_{NEGATIVE}$), where $p_{YES} = \frac{(TP+FP)*(TP+TN)}{N^2}$ and $p_{NO} = \frac{(FN+TN)*(FP+TN)}{N^2}$. The range of values goes from -1 to 1 , where 1 is the perfect classifier, 0 is a random classifier, and less than 0 means that the classifier is worse than a random one. This metric is claimed to be more robust than the previous ones that do not take into account all values in the confusion matrix, but this is the precise reason why I decided to do not use it: this metric can be useful with a balanced dataset but in my scenario, as I said before, TN is much higher than the other values in the matrix, and as this metric uses this value the obtained values will not be reliable.

So I selected the F-score as my metric because it takes into account the essential values for a prediction task in an imbalanced dataset. After all, it has a simple calculation and interpretation with a range of values between 0 and 1 , and because it is widely used. Regarding this last point, I must name another widely used metric: the area under the ROC (Receiver operating characteristic). Binary classification models use a threshold to decide if their output is positive or negative. The ROC is the curve of the accuracy against the FPR at various levels of

this threshold. The metric is the area under that curve and takes values between 0 and 1 (though values under 0.5 are obtained from models worse than a random output). This metric's main problem for my scenario is that it uses the TP count to produce very high values even for bad prediction models.

These metrics are useful for evaluating the prediction models as ML models, and in fact, I use them to select or compare different models, but they may not be so useful to evaluate the models as part of a whole prediction system. When I evaluate a single sample, I am checking the output of a model at a specific time based on the input, but as the prediction system is intended to work in real time, the input will be the same as long the same set of events fall into the observation window, but the target event might not be in the prediction window. This situation produces a misclassification from the point of view of ML evaluation, but from the prediction view, the output should be considered correct if, at some time, the model generated the alert. This process is detailed in Chapter 7.

Having a large prediction window might increase the number of correct predictions, but it has a problem of interpretation when these models run in real time or, in general, when the predictions are overlapped. In many scenarios, this is not a problem, e.g., a model that every day at 00:00 AM predicts if there will be a failure in the next 24 hours. Nevertheless, in my scenario, I will calculate the predictions each second, and that means unless the prediction window is one second, the predictions will overlap. To illustrate the problem, I use Figure 5.2 which shows a hypothetical prediction system output at three different moments. At time t_1 , the system produces a prediction for the event e , saying that that event will happen during the prediction window, which is from t_{1p} to $t_{1p.end}$. In the second situation, sometime after, at time t_2 , the system may say that the same event will happen on a new prediction window, from t_{2p} to $t_{2p.end}$. Finally, at time t_3 , the system produces a negative prediction, saying that the event will not happen on the prediction window, from t_{3p} to $t_{3p.end}$. At time t' , I have 3 overlapping outputs, 2 of them saying that the event will happen and one saying it will not. As there is no way to decide which output is correct before seeing what happens on t' , the predictions are contradicting. This is because the predictions overlap, so the simplest way to avoid it is to use a prediction window equal or less to the time between predictions (1 second in my scenario). As the prediction window is reduced to my data's minimum granularity, I call this kind of window schema Instantaneous Prediction.

Figure 5.2: Prediction windows overlapping

To ensure that the model's evaluation is robust, I use a 5 K-folds validation (more details of the technique in Appendix B). Ideally, I would use a 10 k-folds, but some of the events have a reduced number of positive samples, and that would produce some fold with few or none positive samples, making it impossible to calculate some of the metrics. For this same reason, I must filter the events that will be modeled, keeping only those with at least 10 samples. Even so, I can not use the standard K-fold algorithm, but I must ensure that all of them have the same ratio of positive and negative samples when I create the dataset partitions. If not, It can happen; some partition does not have positive samples at all, causing some of the metrics to evaluate the models fail. E.g., if I try to calculate the recall with a test partition with no positive samples and the number of False Negative happens to be also zero, the formula will be $recall = \frac{TP}{TP+FN} = \frac{0}{0}$ which is indeterminate. Nevertheless, this does not ensure a stable evaluation in many models. Those events with few positive samples or with few relevant samples may have significantly different scores depending on how the fold partitions are done. To mitigate this effect, when I need a more stable measure of the models' quality, I repeat the K Folds validation 10 times and calculate their average score as the final evaluation.

5.4 Model specification

The models I use in this process are supervised ML algorithms. This means that I know if some sample is positive or not, and I can contrast that information with

the model's output. In other words, I will use labeled samples to train the model.

As stated in Chapter 3, I choose the Spark MLlib library as the main framework to create the models, so I use all the available classification models in that library except for the perceptron (all of them detailed in Appendix B):

- Decision Tree
- Logistic Regression
- Random Forest
- Gradient Boosted Trees
- SVM
- Naive Bayes

All those algorithms have a limited set of hyperparameters to tune, and in most cases, tuning must be done empirically. As all of them are implemented in the same framework, I can perform different typical training time or memory consumption evaluations. Also, I can use a dataset in a format usable by all of them.

I decided to do not to use the perceptron algorithm because it is the most time consuming algorithm to train, and study the number of possible neuron layer settings that will deviate the objective line of this contribution.

The models' input will have a standard format, but the information contained may vary depending on the model. The format is a labeled vector, which means a numeric vector with a discrete label. In this case, the label can only have two values, 0 or 1, negative or positive. This label depends on the target event of the model. Each sample corresponds to a different instance of time, although the information on which time corresponds to each sample is lost when the windowing process is applied to the raw data. Using the terms defined Section 5.2, the label will be positive if the event occurs during the prediction window associated with the sample time.

In the experiments I have done, I have had to apply the windowing to the data on many occasions, so I have tried to make the process as efficient as possible. The way to do this has been using a dataset consisting of two fields: an instant of time and the events that happen at that instant. This way, I only must process each time instant once, which means that the windowing algorithm has a complexity

of $O(n)$, where n is the number of unique instants of time in the data. The algorithm relies on the use of queues to emulate the temporal offset. The pseudocode is shown in Algorithm 1. A sample is generated for every instant of time. To do this, I go through the dataset iteratively. I initialize the algorithm with a vector of zeros of size equal to the number of features that the model will have. In each iteration process, an instant of time and its associated events, I add the events to the end of the queue representing the prediction window. I check if the date of the first item in the queue is less than the time of the iteration event minus the size of the prediction window. If so, I take the event out of the queue and add it to the lead queue. I repeat this until no more events are coming out of the queue. The same is done similarly in the lead time queue and the observation window queue (events leaving the prediction window to enter the lead time, those leaving lead time to enter the observation window, and those leaving the observation window are discarded). With the caveat that each time an event enters, the observation window increases the corresponding value of the feature vector by one; every time an event comes out, it decreases its position in the vector by one. In this way, the vector maintains the representation of the observation window. The label associated with the vector is calculated by checking whether the target event is inside the prediction window. This algorithm takes around $400ms$ to perform the full dataset windowing on Section 3.3.1. This time is not affected by varying the parameters of the window scheme, as it only depends on the number of instants of time.

Algorithm 1: Windowing process: input generation

```

Data: target_event  $\leftarrow$  event to label
t_prediction  $\leftarrow$  size of the prediction window in seconds
t_lead  $\leftarrow$  lead time in seconds
t_observation  $\leftarrow$  size of the observation window in seconds
dataset  $\leftarrow$  sorted collection of time and events

features = zeros
observation = queue()
lead = queue()
prediction = queue()
for event in dataset do
    for index in event.indexes do
        | prediction  $\leftarrow$  insert (event.date, index)
    while prediction.peek().date < event.date-t_prediction do
        | lead  $\leftarrow$  insert prediction.pop()
    while lead.peek().date < event.date-t_prediction-t_lead do
        | zeros[lead.peek().index] +=1
        | observation  $\leftarrow$  insert lead.pop()
    while observation.peek().date <
        event.date-t_prediction-t_lead-t_observation do
        | zeros[lead.peek().index] -=1
        | observation.pop()
    | label  $\leftarrow$  target_event in window yield (features, label)

```

5.5 Methodology proposal

The core of this chapter's contribution is a generic methodology for obtaining event prediction models in datasets such as that described in Chapter 4. This methodology is designed in a generalist way capable of adapting to the data. A noteworthy feature is that it is not only a theoretical sequence of steps but, apart from designing it, I have applied it on a real dataset, obtained from the production environment of a company. This implies that the results presented here are not ideal and some of the steps will not seem very effective, although all of them are necessary. Another difference in working with real data and not with a theoretical process is that proposals must be achievable, and decisions must be made regarding compromises between the results and their difficulty in obtaining. For example, the technique that was named earlier that I use to validate models is a K-Folds cross-validation. In this type of validation, the higher the number of partitions

made the more robust the evaluation of the model. However, it is important to consider what the data working with is; in my scenario, there are events with only 10 positive samples, this means that, although theoretically using 20 partitions is better than using only 5 as I do, for these rare events would mean that some of the partitions made would not have positive samples and therefore the evaluation would not be well defined.

Another issue I have considered in proposing this methodology is the compromise between time and results. I have made several decisions that aim to make this methodology applicable in a limited execution environment, such as that presented in "environment A." In most ML projects the goal is to achieve a single model, in this case, the goal is one model for each type of event, and this causes any temporary cost that involves a decision that is made, must be multiplied by the number of events in the dataset. Therefore, although some processes are inevitably slow, I have look to have the decisions made to consider this tradeoff.

The methodology consists of five sequential steps. Each of these steps involves resolving one of the usual issues in an ML process (e.g. Algorithm Selection or Variable Reduction) taking into account the characteristics of the problem. Initially, all events are attempted to be modeled in the same way, but each step of the methodology involves a decision about the model configuration that affects successive steps and ends with individualized modeling for each event type. Decisions made in each of the steps can be automated based on a given configuration, allowing all steps to be assembled in a single process. The fact that the methodology is generically adapted to a dataset and can be automated fulfills one of the objectives proposed in Chapter 3: it must be an achievable process without human intervention and the need for knowledge in the domain or the technique.

The sequence of steps of the methodology is shown in Figure 5.3 and a brief description of each of them is as follows:

- Step 1. Preliminary evaluation: in this step, all events will be modeled using a common windowing scheme and the same algorithm with equal parameters. The goal is to achieve a basis with which to compare the progress of the next steps.
- Step 2. Optimization of the window scheme: This step optimizes each of the events the window scheme uses in the model. This involves deciding the size of the model observation window and the lead time, the time separating the observation window from the validity moment of the model prediction.

- Step 3. Algorithm selection: For each of the events, the algorithm and combination of hyperparameters that work best are searched.
- Step 4. Feature reduction: Reduces the number of events that make up the model input for each event, so that only those that are related to the occurrence of the modeled event are maintained.
- Step 5. Sampling: To evaluate the true performance of a model in an unbalanced set, techniques are applied that modify the training set to balance the number of positive and negative samples.

The following sections of this chapter will discuss in detail how each of these steps works and the results I have obtained when applying them in my scenario.

Figure 5.3: Methodology steps

5.6 Methodology in detail

5.6.1 Classic windowing

This is step number 0 because it does not truly belong to the methodology. Even so, I included it because it can be useful to compare the models obtained with the Windowed Prediction and the Instantaneous Prediction. It is foreseeable that these models will have better F-score than the ones obtained with Instantaneous Prediction, it is intuitive that it would be easier to guess that something will hap-

pen in a window of five minutes than in a single second, but this is the same reason that makes impossible to compare the results directly: the interpretation of the models' output is different and, in consequence, the score of the model is different too. Nevertheless, it can be useful to know how many events are predictable with each of the approaches, and even in some cases, it may be interesting to analyze why it is possible to generate a valid model with one approach but not with the other.

The window schema is an observation window of 5 minutes followed by a prediction window of also 5 minutes without a lead time between them. This window size is decided according to the median value of the time difference between 4 events in the dataset. In other words, is the window in which, in the median, there will be four events. This is, of course, an arbitrary value with no justification. This will be fixed in step 2. I train a Random Forest model for each event in the dataset with more than ten samples. The Random Forest hyper parameters are selected with a grid search of all possible combinations, performed for three random events of the first quartile of most frequent event rank.

Input

This step uses the whole dataset to generate the training data. There are 168 events with at least 10 samples. The features vectors will be the same for all the events, only the label associated with the vector changes. So it is enough to calculate one time all the possible observation windows in the dataset, and save which events happen in their corresponding prediction window to, later on, convert those lists in the label by checking if some individual event is in the list or not.

Going back to Algorithm 1, the sample's feature vector is created by taking a snapshot of the system's past and checking what happened during the observation window, so the current time is the end of the observation window. When working with offline data, there is no such *current time*, and it is possible to take a snapshot of the system at any time (inside the dataset period). The extreme is to take a snapshot of each second in the dataset, which will produce 17.8M samples, but most of them will be useless, provided that the events are sporadic and, in consequence, the observation window will be empty. To get the maximum number of different samples, I first obtain the list of all different event times in the dataset, which are 10661. Then I create samples assuming those times correspond to 4 possible situations, as shown in Figure 5.4: the start and end of the observation window, and the start and end of the prediction window. This process may lead to duplicated times (e.g., two event times differing precisely the size of the observation window), so I create a list without duplicates containing all possible

times, and then I calculated the samples. Thus, for each event time in the dataset, I create a maximum of 4 time samples, 42644, but the actual number of samples is 40702.

The windowing process is performed with low level code, and it lasts a few seconds even in Section 3.3.1.

Figure 5.4: Classic windowing schema

Process

In this step, I use the Random Forest algorithm with the default parameter configuration, that is (parameters are explained in detail in Appendix B):

- FeatureSubsetStrategy: sqrt
- Strategy: gini
- MaxDepth: unlimited
- NumTrees: 100

With that parameter combination, I trained a Random Forest for each event, and I evaluate them with the F-score in a 5 k-fold, so I trained 840 models. The training process lasted 95 minutes.

Results

In the following Figure 5.5, I have represented the evaluations of the models of each of the events. The bars are sorted in ascending order of event indexing. The height of each bar represents the F-Score obtained by the model when evaluated with a 5-k-folds validation. Among the first 20 events, the score fluctuates quite a bit, probably because it is much more frequent than the rest (the frequency of events has a trend that follows a power curve). The samples' variability will be substantial since no preprocessing of the data is done, and much noise is introduced (e.g., events that are unrelated and only add variability to the input). The central part of the graph shows outstanding models with scores close to the maximum. Finally, on the chart's right side, the scores for the last quarter of events have a declining trend. Events with very few samples are not able to generate an efficient model.

Figure 5.5: Step 0: model scores

The average score is 0.756, the median 0.819, and 115 of the 168 (68%) modeled events to get a score of at least 0.7. These measures are significantly better than I get when applying a window scheme for instant prediction. However, as I have previously argued, this type of window prediction is problematic to apply in continuous or real-time prediction, although it can be useful in other applications.

Because of the way samples are generated, many of the events have more positive samples than their actual frequency. Figure 5.6 shows the percentage of positive samples for each of the events. From this figure, I get an interesting fact, some of the events with a lower frequency have more positive samples than the

events more frequently, of this, I can infer that these rare events happen in periods of high activity on the network, which makes them reflected in many prediction windows.

Figure 5.6: Step 0: the ratio of positive samples

To conclude this section, I talk about training times. In general, a machine learning model's training time is proportional to the size of the dataset (in most cases, linearly proportional, but in some cases, it can be a quadratic or larger ratio). On average, each model takes 1.5 minutes to train and validate, accumulating a total time of 4 hours in environment B.

5.6.2 Instantaneous prediction

This is the first step in the methodology, and it serves as a base to compare the gains of the further steps. Similarly to the previous step, I train a Random Forest for each model with at least 10 samples and score them with F-score and a 5 K-folds validation. In this case, the window schema is an observation window of 5 minutes and a prediction window of one second separated by 60 seconds, in other words, an instantaneous prediction with a lead time of 1 minute. This step will not discard any event, but it is useful to get an idea of which events can be modeled.

Input

This step's input only needs the event index of each of the events that will be modeled (those with at least 10 samples). I store this list into a JSON file as a list of entries with all the information needed to generate an event model. With

the next steps' results, more info will be added to each entry of the list. At this point, the file has this schema:

```
[
  {
    event: #Event_name_1,
    index:0
  },
  {
    event: #Event_name_2,
    index: 1
  },
  ...
]
```

All the models use the same window schema, so the features vectors in the training data are valid to any model by only change the label. The windowing process is more straightforward than classic window schema, as the prediction window is reduced to the minimum possible size (1 second), it is only needed to generate the samples when there is an occurrence of one or more events. Consequently, there will be as many samples as event times in the dataset, which is 10661.

Process

The process is the same that described in the previous step. The most significant difference is the number of samples, around 25% of the previous number. In consequence, the ratio of positives samples over the total samples decrease.

Figure 5.7 shows the event distribution according to the ratio of positive samples. In the worst case, the event only has 10 positive samples over a total of 10661 samples, which is only a 0.1% of positives. From these limits, situations will be complicated to obtain a feasible model, and I do not expect to achieve that unless in the case of simple events (e.g., an event A that is always preceded by an event B). In the best case, the event indexed as 0, the positive sample ratio is 10%, which is still a small number.

Figure 5.7: Step 1: positive samples ratio

The process is exactly the same as explained in the previous step, provided that the data format is the same (a dataset of labeled vectors), and the algorithm also has the same settings. The only difference is the input data. This time, the dataset has only 10k samples, so the training process is faster, taking 32 minutes.

Results

Figure 5.8 shows the score distribution of the trained models. As it was expected, the average score is lower than in Step 0, but as it was said, those numbers can not be directly compared.

Figure 5.8: Step 1: models score distribution

Figure 5.9: Step 1: relation between models' scores and samples

It is interesting to compare the model scores and the ratio of positive samples, as I show in Figure 5.9a. Each event has a point, with model scores in the vertical axis and the positive sample ratio in the horizontal one. There is no direct relationship between those two values, which reveals that the difficulty predicting an event does not depend on the number of positive samples and that there are *simple* events that can be modeled even with a low number of samples. On the other hand, some events with a higher number of positive samples obtain bad results. This implies that even with many samples, there are no event patterns in the observation window, or at least the algorithm did not find them, which allows predicting those events. Retrieving the concept of predictable sample ratio introduced in Chapter 4, Figure 5.9b, similar to the previous one, shows the model score compared to the percentage of predictable samples. In this case, It is possible to a certain relationship between models with a low, predictable sample ratio and bad scores.

Finally on the next figure, Figure 5.10, I represent the distribution of the model scores on a boxplot and a violin plot to visually interpret the results. The average score is 0.209, the median 0.134, and the maximum 0.796.

Figure 5.10: Step 1: Boxplot and violin plot model scores

5.6.3 Observation window size optimization

The previous step just provided a scoring base with which to compare the improvements that each of the following steps adds, therefore this is the first step that improves as such the models obtained. The main idea is that different events may need different observation windows, and each of them must be analyzed separately but automatically. Based on the work presented in [10], I use a genetic algorithm that optimizes the observation window's size and lead time for an event based on the performance of a Random Forest model that uses training data with that window scheme.

The algorithm I use is the same as in the published job, but I have reimplemented it in Python and SkLearn. This algorithm is a heuristic that needs to train and evaluate a large number of models, but its level of parallelization is limited by the size of the population used, making a distributed system like Spark unable to make the most of its computing capacity. One solution would be to run the algorithm for multiple events in parallel, but this requires an excessive amount of memory since all data from each event must be distributed in the cluster. To avoid this limitation, the algorithm is implemented to run on a single machine, therefore, a machine only has access to the data of an event at the same time allowing efficient use of available memory.

The algorithm was executed in environment Section 3.3.2. Each of the computers works in isolation and dumps the results of each execution of the algorithm in MLFlow. The total processing time was 16 hours, an average of 42 minutes per

event (parallelized in the 8 machines).

Input

This time the window scheme of each of the events will be different and will, in fact, vary during the execution of the algorithm. Therefore, the input will not be a dataset representing the data with a specific window schema, but the raw time and event dataset. Besides, the JSON file with the list of events and their corresponding index is provided.

Process

The algorithm configuration is the same used in [10], with the following exceptions. The population size is incremented from 20 to 50 to allow a more exhaustive space search. The K-folds validation changes from 3 folds to 5 folds to provide more stable evaluations. Moreover, finally, I added 2 stop criteria to allow early finish instead of performing the maximum number of generations (50). The first one check if the maximum F-score observed in the first iteration is zero, which means that any of the models trained could predict just a single positive prediction. The second one looks for early convergence: if both the typical deviation of the window size and F-score of the best individuals of the last 5 iterations (including current generation), relative the values of the last generation, are less than 0.01, the algorithm stops. In the case that all of the individuals in the generation achieve an F-score of 0, the algorithm also stops. The pseudo code of this stop criteria is Algorithm 2.

Algorithm 2: Stop criteria: early convergence in genetic algorithm

Data: score_history $\leftarrow$ F-scores of best model in last 5 generations
 window_history $\leftarrow$ Window size of best model in last 5 generations
 score $\leftarrow$ F-score of best model in current generation
 window $\leftarrow$ Window size of best model in current generation
 std_score = std(score_history / score)
 std_window = std(window_history / window)
if *std_score* < 0.01 **and** *std_window* < 0.01 **then**
 stop optimization

Each machine runs isolated the genetic algorithm for an event at a time. The event to process is decided by querying the results stored in MLFlow, choosing one of those that are not finalized or is in process. As soon as an event is chosen to process, it is marked as "in process." At the end of each iteration of the algorithm, the generated results are stored in MLFlow. When the algorithm finishes, the result is stored, and the event is marked as finished.

This is the step that most computing needs. For each of the events, the genetic algorithm will be executed, which can have up to 50 iterations. In each of those iterations, 50 window schemes are tested and validated with a 5 K-fold. This is a maximum of $50 * 50 * 5 * 180 = 2,250,000$ trained models. Thanks to the stop conditions, this number is significantly reduced because the algorithm converges early, It does not need to perform all the iterations, and in the actual case, only 211650 models were trained. The execution of the algorithm in the scenario Section 3.3.2 took 16 hours.

Results

Figure 5.11 shows a first view of the results. In the chart, each of the events is represented sorted by index with the score that its model gets. For some events, the algorithm has failed to find any window size with which a model can be generated to predict any event occurrence, that is, the models with F-Score equal to zero. Most of these events are those with higher indices, i.e., those with the least samples, but some have low indices such as the '#11'. From these events, I cannot generate a model that works minimally well. Therefore I discard them, and they will not be used in further steps. Of the 168 models, 95 passes this filter (discarding 73).

Figure 5.11: Step 2: Window optimization raw models scores

Figure 5.12 shows the results again but having discarded events that do not pass the filter. Events are still sorted in descending order by the number of samples, just as seen in Step 1. There is no direct relationship between the number of samples in an event and the model's result.

Figure 5.12: Step 2: Window optimization filtered models scores

To be able to see in more detail, Figure 5.13 includes several boxplots with the distributions of the different variables. Figure 5.13a is the distribution of the F-score models before and after the genetic algorithm is executed. The average score is increased from 0.209 to 0.422, the median from 0.134 to 0.364, and a maximum of 0.796 to 0.853. Figure 5.13b shows the distribution of the time in minutes that the optimization took to run, on average, each event takes an hour and a half to optimize its window scheme.

Figure 5.13c and Figure 5.13d charts are the lead time and window size distributions, respectively. The first thing that can be noticed is that the 5 minutes used in step 1, are clearly insufficient since the average size of the optimized windows is 50 minutes, which is ten times more. Window sizes range from a few minutes to 6 hours. For the lead time, most events are set to the minimum allowed (60 seconds), but others grow up to 30 minutes.

Finally, from this step, I include the following graph depicting the improvement of each event's score, adding in each case a new step. Each of the lines represents a model; each vertical is one of the steps of this methodology. I have included a boxplot of the model scores improvement between the two steps on the right side of the image.

Figure 5.13: Boxplots of the model variables distributions

Figure 5.14: Step 2: Model scores evolution

5.6.4 Model selection

Once I know the best observation window scheme and lead time for each event, I must also find which algorithm works best and parameters. The library Scikit Learn offers a wide range of algorithms for generating classification models; however, Spark's MLlib library is more limited and only includes the algorithms that are explained in Appendix B. As the models I get after applying this methodology are used in another process that uses Spark as the main framework, I limit my search for algorithms to those implemented in MLlib. If this limitation does not exist or if different algorithms should be included, the process will remain the same, only needing to configure the new algorithms in the search grid.

Input

Since this step, the input file is more complex because, for each of the events, I have calculated a different window scheme, besides it only includes the events that passed the filter specified in the previous step:

```
[
  {
    event: #Event_name_1,
    index:0,
    observationWindow: #Window size in seconds,
    leadTime: #Lead time in seconds
  },
  {
    event: #Event_name_2,
    index: 1,
    observationWindow: #Window size in seconds,
    leadTime: #Lead time in seconds
  },
  ...
]
```


Process

Some of the algorithms used have parameters with continuous values. For those cases, a standard grid search could be computationally expensive; instead, I run a Random Search [111]. Although it will not yield the optimum parameter set for each algorithm, it will provide a good-enough solution in a fraction of the time that runs the grid search would take. To run this kind of search, I must define the range of the parameters, with the difference that I can set distribution of values instead of a closed set of values and the maximum number of search iterations. This number of iterations is the variable that controls the balance between the expected quality of the results and the computation time needed. After some test runs, I decided to set this value to 100 iterations. The tests consist of trying the random search with 10 events selected randomly and varying the number of iterations from 50 to 500.

Figure 5.15 shows boxplot charts of the distributions of scores and computation time depending on the number of iterations. The average score does not change significantly, although it has a positive trend with the number of iterations. On the other hand, the time is directly proportional to the number of iterations. So the decision is a trade off between the time and the expected quality of the solution. I selected 100, but if some scenario needs more accurate models or the computation power is higher, this number can be increased.

Figure 5.15: Step 3: Random Search iterations

However, I can check if the search for some event went bad, just comparing the results with the ones from the previous step. If there is the case that some event obtains a model with worse than the produced in Step 2, I run again the search for

that event increasing the number of iterations to 250. This could be repeated until I get a better model, but I am only doing it once, because the Machine Learning models are statistical models created with a pseudo random process, and it might have happened that the model in Step 2 was i.e., overfitted, and its score was higher than it supposed to be.

In the case of tree based algorithms, I use a standard grid search. As in the next step, I will reduce the number of features used in the models. It makes no sense to optimize the FeatureSubsetStrategy (the proportion of features used to decide on the trees), and I set this parameter to use all features in each decision. In the decision tree case, the number of possible configurations is only 50, so it is acceptable to do a standard grid search. For RandomForest, the number of trees used is not a sensible parameter, and the search can be coarse, so I only try 8 configurations. Finally, GradientBoostedTree is with a significant difference the most time consuming algorithm to train, so I decided to use a limited grid search.

The following Table 5.2 contains the ranges or values distributions that I have tested in each algorithm and parameter, the explanation of the algorithms and their parameters can be found in Appendix B:

Algorithm	Parameter	Values
Random Forest	FeatureSubsetStrategy	all
	Strategy	$\in \{\text{gini}, \text{entropy}\}$
	MaxDepth	30
	NumTrees	$\in [50, 100, 150, 200]$
Decision Tree	FeatureSubsetStrategy	all
	Strategy	$\in \{\text{gini}, \text{entropy}\}$
	MaxDepth	$\in [5 - 30]$
Gradient Boosted Tree	FeatureSubsetStrategy	all
	MaxDepth	30
	numIterations	$\in [20, 30, 40, 50]$
	StepSize	$[0 - 1]$
Logistic Regression	regParam	exponential $\exp(-100x)$
	elasticNetParam	uniform in $[0, 1]$
	maxIter	logaritmik uniform in $[10, 100000]$
Support Vector Machine	regParam	exponential $\exp(-100x)$
	maxIter	logaritmik uniform in $[10, 100000]$
Naive Bayes	Smoothing	uniform in $[0, 1]$

Table 5.2: Algorithm hyperparameters configurations

Preliminary results reveal that the Logistic Regression models were quite bad at this task, and the training time is the highest between all the algorithms. Figure 5.16 shows the results of train 10 random events with all the algorithms. On the left, each event's score with each algorithm, and the training times on the right side. For some events, the Logistic Regression models score an F-score of 0.0, yet it was the slowest algorithm. In all cases, the score of this algorithm was surpassed by the others, so I decided not to include it on the last search.

Figure 5.16: Step 3: logistic regression results

Results

Figure 5.17 shows the aggregate distributions of model scores and training time for each of the algorithms. From Figure 5.17a, the algorithm ranking according to their average F-score is Gradient Boosted Tree, Random Forest, Decision Tree, SVM, and finally, with a significant difference, Naive Bayes. According to the times, as shown in Figure 5.17b, the fastest is the Decision Tree and Naive Bayes (average of 7 and 9 seconds respectively), followed by Random Forest (6 minutes), SVM (17 minutes), and finally, Gradient Boosted Tree (51 minutes). So, although the models' scores are similar, the training time oscillates widely depending on the algorithm.

Figure 5.17: Hyperparameters Random Search results

A closer view of the results, analyzing the algorithm behavior for each event, reveals that, as expected, the best algorithm is not the same for all events. Nevertheless, in many cases, several models have similar scores, like the ones shown in Table 5.3 (The full version of this table and a similar one with the training times can be found in Appendix C). In those examples, the maximum score differs little from the next one. For example, for event 38, the best algorithm is GBT with an F-score of 0.7153, and Random Forest has 0.7118, but in training time, they last 2606 and 211 seconds, respectively; that is a ratio of ten to one for a score improvement of 0.035 in the F-score. In that kind of case, I choose to keep the fastest method. Specifically, for each event, I select the fastest algorithm in which its F-score is equal or greater than the maximum F-score for that event minus 0.01.

event	DT	GBT	NB	RF	SVM
26	0.7925	0.7936	0.7371	0.7895	0.6635
55	0.4502	0.4252	0.1192	0.4384	0.1053
128	0.3741	0.38	0.05924	0.38	0.3411

Table 5.3: Step 3: detailed view of different algorithm model scores

In Table 5.4 I included the counts of how many times each algorithm was chosen. It seems that algorithms based on Decision Tress are the most effective with this scenario. Interestingly, the Decision Tree algorithm was chosen 19 times, one more than the Random Forest, that may suggest that those events are easier to predict and do not require a complex model.

Algorithm	Max with tolerance	Max
GBT	48	55
DT	19	17
RF	18	15
SVM	5	3
NB	1	1

Table 5.4: Step 3: Selected algorithm count

To end this section, I include the evolution chart with this new step in Figure 5.18. For some events, the score has lowered a bit. Those could happen provided that a single correct sample may vary the evaluation widely when dealing with events with few samples. The average F-score has risen from 0.422 to 0.450. Run this step in Section 3.3.1 took a total of 14 hours.

Figure 5.18: Step 3: Model scores evolution

5.6.5 Feature selection

I have already obtained some models with a windowing scheme suitable for the event at this stage, and I have selected which algorithm best suits the event. The goal of this step is to reduce the number of features of the models. Hopefully, just as not all events need an observation window of the same size, not all events will be influenced by all events in the dataset, but some will be related in their occurrences while others will be totally strange. Most classification models can be examined to extract what is known as the influence of variables; this is a numeric value that represents the weight that a feature has in the model output. This value does not have to be narrowed, but it is proportional to the variable's influence. That is, if the value is high, the weight of the variable is high as well. In this case, features or variables are the occurrences of different events in the observation window. The importance can be understood as how much it affects whether or not an event appears in the observation window for the modeled event to appear in the prediction window.

Once this vector of influences has been extracted for each of the models, the next step is to eliminate the ones that are less important in an iterative way without affecting the model's performance. Some algorithms, such as Random Forest, can ignore variables that do not provide information; however, other types of algorithms may be harmed by the presence of variables that do not provide information. Therefore, complex models are expected to maintain their score, but simple models can even improve by applying this step.

Input

I added the selected algorithm and configuration of each event to the configuration file used in previous steps:

```
[
  {
    event: #Event_name_1,
    index:0,
    observationWindow: #Window size in seconds,
    leadTime: #Lead time in seconds,
    algorithm: #Algorithm to train this event model
    parameters: { #Algorithm params
      param1: #param1 value,
      ...
    }
  },
  {
    event: #Event_name_2,
    index: 1,
    observationWindow: #Window size in seconds,
    leadTime: #Lead time in seconds,
    algorithm: #Algorithm to train this event model
    parameters: { #Algorithm params
      param1: #param1 value,
      ...
    }
  },
  ...
]
```

Process

The process is relatively simple because ScikitLearn provides methods to directly extract the vector of importance and algorithms to reduce the number of features automatically. However, it is interesting to analyze the vector of importance itself. Figure 5.19 shows the charts for models of some different events. Ideally, these bar graphs would show, sorted from highest to lowest, the importance of each of the model variables, but the number of features at this point is equal to the number of different events in the dataset. Therefore they are filtered to show only the 30 most important variables of each of the models. The line, which corresponds to each of the graphs' right axis, is the CDF of these variables so that on the right side, it shows what percentage of total importance these 30 features represent. The color of each of the bars is decided according to the type of machine in which the corresponding event occurs, and the color of the line depends on the type of machine on which the modeled event occurs. Also, the bar that corresponds to the feature that measures the occurrence of the event itself being modeled (i.e., the same event you are trying to predict appears in the observation window) has been highlighted with a black border.

Figure 5.19: Boxplot of the model variables distributions

The first conclusions drawn are that the importance of variables declines rapidly, indicating that only a small set of events actually produce the patterns used to predict each event's occurrence. Another quick observation is that events are more influenced by other events occurring on machines of the same type. This makes sense because events that occur on switches will have to do with the network level,

while virtual machines can reach the application level and a networking event is more likely to generate other network events, even if it might trigger on an application-level event. For these figures, I have chosen events covering the other types of machines or observed distributions. Nevertheless, still, some general measurements can be extracted from all models (on average):

- The 30 first features, in order of importance, represent 96% of the total importance of each model
- Among the first 30 features, the 60% correspond to events that occur on the same type of machine as the modeled event
- In 24% of cases, the modeled event itself was among the 30 features. This may indicate that the event occurs in bursts.

The process of adjusting the number of features required for each model is straightforward: one of the input variables is removed, usually, the less important ones, the model is re-trained, and the evaluation is compared. This is repeated until all input variables have been removed. With the series of evaluations with the different numbers of variables, I look for the model that gave the best result and saved the list of chosen variables. This list of variables can be interpreted as new indexing of the events that will be specified by each of the models.

As I mention in other steps, the model training is a pseudorandom process, and with an unbalanced dataset, small variations in the training could lead to significant differences in the model evaluation. In this step, I am evaluating the models with a different number of features, and I need the evaluation to be as much stable as possible, so instead of only using the 5 K-Folds cross validation, I perform this cross validation 5 times take the average result as the output. This technique obtains more stable solutions, which means training the model with the same parameters several times, produces fewer variant evaluations scores -but this is still a random process, so there will be variance in the evaluation-, but with a tremendous computational cost. A comparison between to apply or not, this repeated cross validation is represented in Figure 5.20. These charts represent the model evaluation of a single event, depending on the number of features used to train the model. The standard K folds are clearly noisier and may lead to selecting a too high or low number of features.

Figure 5.20: Evaluation of Decision Tree model of event #8 depending on the number of features selected

To reduce the number of evaluations performed, I divided the process into two stages. In the first one, I do a coarse search removing 50 features in each iteration and keeping the best solution found. The second one is a fine search in which I perform the same search but removing only one feature per iteration and starting with several features equal to the best found in the first stage.

To perform this kind of iterative removal of features, I should be able to extract a list with the feature importance or the coefficient of each feature. Unfortunately, this can not be done with Naive Bayes, so the events that used this algorithm are omitted in this step (which are only 2 of the 100 selected in step 2).

Something particular of this kind of dataset is that when features are discarded, the dataset itself changes. This is because the dataset is built as a set of observations of the events apparitions, but if some event is no longer relevant for the model, that event should be ignored when the windowing is performed. If not, they will produce duplicated or useless samples, which could negatively impact the model training. However, redo the windowing process is a costly process, so it is only done after each stage.

Results

Most of the event had a behavior similar to the shown in Figure 5.20, which is a logarithmic shape. That means the first group of features have an intense influence on the model score, but the others contribute significantly less, and the chart trend

is almost zero. Even with the repeated K Folds, as I keep the maximum value, on many occasions, the algorithm probably selects a higher number of features than it was necessary. Nevertheless, I prefer to be conservative in this step because there is a big risk of discarding too many features with more aggressive approaches.

Before this step, the number of all events' features was 548 (all possible events in the dataset). This number is reduced to 135 on average, 86 in the median after this stage. However, examining the distribution of the number of features selected in Figure 5.21, there is a dominant group with a low number of features, but some models (e.g. the group around 200) probably select too many features because of the noisy evaluations even with the Repeated K-Folds.

Figure 5.21: Distribution of the number of features selected

This step is the one that less impact has on the average score of the models. For some events, it even deteriorates it. However, it is a necessary one, in the first place because it reduces the training time, but most importantly, because it reduces the influence of other events in the model. In Chapter 6, these models are used and combined with others feeding others' output. Those models' quality is not always good and making that a model depends on fewer events will reduce the chances to be influenced by a bad one.

This step was executed in Section 3.3.2, running the same program, which worked with one event at a time in each machine, and it lasted 6.3 hours. The average F score of the models passes from 0.21 in Step 3, to 0.24 after this step. The evolution of the individual events is plotted in Figure 5.22.

This step was executed in Section 3.3.2, running the same program, which worked with one event at a time in each machine, and it lasted 6.3 hours. The

models' average F score passes from 0.21 in Step 3 to 0.24 after this step. The evolution of individual events is plotted in Figure 5.22.

Figure 5.22: Step 4: Model scores evolution

5.6.6 Sampling methods

Sampling is a technique that is frequently used when working with unbalanced datasets. The idea in the case of a dataset with two classes is simple, one of the two classes is dominant, usually the negative one, with several samples far higher than the other. The goal is to get a set of samples where the classes are balanced. This can be achieved either by removing negative samples, either by adding positive samples or with a combination of both. When samples are deleted, it is called undersampling: repeated or similar samples are attempted to be deleted to maintain the information contained in the dataset. When adding samples, it is called oversampling: there are two ways to do this either by duplicating existing samples or by creating new synthetic samples by combining existing ones.

In this step, I test several sampling algorithms with each of the events. It is to be expected that different events and different models can perform differently with the different sampling options, and therefore I try all of them and select the one that works best in each case. The techniques I use are as follows:

- Oversampling
 - Random oversampling
 - SMOTE oversampling [85]
 - ADASYN oversampling [112]

- Undersampling
 - Random undersampling
 - Cluster Centroids undersampling
 - AllKnn undersampling
- Combination of under- and over- sampling
 - SMOTE ENN
 - SMOTE Tomek

Details about these methods and the library used are described in Appendix B and Appendix A.

Input

This is the last step of this methodology, and the configuration for each of the events is totally unique: specific lead time and observation window, the algorithm with parameters, and set of features used in the model. The features selected in the previous step are coded in the JSON file as two lists of integers: the first is the original index of the event, and the second the new index. By combining both, I create a mask that I apply to all events in the raw dataset, only events in the mask will be used in the windowing process for a particular event.

```
[
  {
    event: #Event_name_1,
    index: 0,
    observationWindow: #Window size in seconds,
    leadTime: #Lead time in seconds,
    algorithm: #Algorithm to train this event model
    parameters: { #Algorithm params
      param1: #param1 value,
      ...
    },
    features: {
      from: [0,2,...],
      to: [0,1,...]
    }
  },
  {
    event: #Event_name_2,
    index: 1,
    observationWindow: #Window size in seconds,
    leadTime: #Lead time in seconds,
    algorithm: #Algorithm to train this event model
    parameters: { #Algorithm params
      param1: #param1 value,
      ...
    },
    features: {
      from: [10,7,...],
      to: [0,1,...]
    }
  },
  ...
]
```

Process

The process is quite similar to other steps. I start with a collection of techniques. In this case, sample methods, to test in each event, I register the model score obtained with each and finally select the best one.

One particularity is that I can not directly use the K Folds method. When some kind of sampling is applied, it should only be applied to the training data; so I create the different partitions of test and train data and apply the sampling only to the training, then I aggregate the score of all partitions with the average.

Results

The results of this step are the final models that I get after applying the whole methodology. The metrics I present in this section approximate how the actual model works, as the actual model will be trained using the entire dataset as a training set.

Sampling algorithms of 3 categories have been tested: undersampling, oversampling, and mix. It may be the case that some models are not able to improve their performance when applying these techniques, in which case, the decision will be that none of them be applied. Table 5.5 on page 80 shows how many times each of the techniques has been chosen. None of the techniques predominates significantly over the others.

Model	count	
None	33	35%
Oversampling	24	25%
Undersampling	21	22%
Mix	17	18%

Table 5.5: Sampling technique selected

The result distribution is shown in Figure 5.23. Although the average goes up from the previous steps, there are still models with shallow metrics. These models will be discarded when the model set is used in any application.

Figure 5.23: Step 5: Boxplot model scores

Finally, Figure 5.24 shows the complete evolution of the models throughout the methodology. A closer look at the last transition reveals that there are models that improve little or nothing, but there are others that have climbs of more than 0.1. Compared to step four the average F-score has risen from 0.44 to 0.47. This step took 8 hours to run in environment b.

Figure 5.24: Step 5: Model scores evolution

5.7 Conclusions

The methodology presented in this chapter is a long and complex process but designed in such a way that it can be executed automatically for a generic dataset

only needing human intervention in the initial configuration. The techniques and steps performed to follow the philosophy of a typical data preprocessing process in machine learning, but with the considerations of a dataset with different characteristics than usual. The most significant difference is that while in a machine learning project, the task is to prepare and train a single model, in my scenario, I have worked with a hundred models, as many as different events. This means that any process that wants to apply I have to repeat it for all models, which can become an unfeasible process, so the decisions that are made in this set of steps always take into account the tradeoff between results and time. Therefore, the models obtained by applying this methodology will not be the best possible, but the best that can be obtained in time according to the project's needs. Most of the processes performed can be easily modified in the case that is needed to perform a more fine-tuned process.

This methodology's steps are presented in a specific order, the right one in my scenario, but it is not an immovable order. For example, the selection of the modeling algorithm might be made after feature reduction. Not only is it possible to change the order, but it is also possible to repeat some of the steps. In my case, for decision tree-based algorithms, I decided to set the parameter that controls the percentage of variables used in each decision at 100%, because then I modify the number of characteristics, and it makes no sense to use a value that is decided based on the maximum number of features. After this feature reduction in step 3, I could have repeated step 2 by modifying this parameter.

I started this methodology with 168 events, all those with at least 10 occurrences in the dataset. Of 95 of those events, I have been able to generate a prediction model. The following Table 5.6 on page 82 shows the evolution of the F-score models at the end of each of the steps. In both medium and median, the value grows with each step, except in step 3. To understand how these metrics are distributed, I include Figure 5.25, where I show a boxplot of each step's results and the individual values of each model in the form of points to appreciate the shape of the distribution.

	Step1	Step2	Step3	Step4	Step5
Mean	0.208694	0.422145	0.450056	0.436250	0.477193
Median	0.134444	0.363878	0.393333	0.358632	0.411532

Table 5.6: Scores evolution

Figure 5.25: Scores distributions

As for the time spent getting these models, most of the steps involve a lot of calculations. The total time spent has been 45 hours, the times of each step and by events are summarized in Table 5.7 on page 83.

	Step1	Step2	Step3	Step4	Step5
Total (hours)	0.5	16	14	6.3	8
By Event (mins)	1	43	42	32	5

Table 5.7: Computation time summary

5.8 Chapter summary

This chapter covers how the network events dataset described in Chapter 4 is treated to create a set of prediction models that can be used to forecast the appearance of events based on an observation of the network in the previous instants.

To create this observation of the behavior of the network in the past, I defined a windowing mechanism which allows codifying a particular instant of time into a numeric vector and a label, based on two parameters: the time that is taken into account, that is the observation window, and the temporal difference of that window's end and the time being codified, that is the lead time. The vector label is a binary value representing the occurrence or not of a particular event at the codified time. The numeric vector is a count of the events produced during the observation window.

Then I described how I create these prediction models using standard Machine Learning techniques, specifically binary classification models. Using this kind of model, I can take advantage of a collection of methods to evaluate the models, most of them based on a simple but effective representation of the model results: the confusion matrix. From all the possible metrics, I select the F Score because it is influenced mainly by the aspect of the model that is interesting for prediction tasks.

Finally, the chapter's core is a proposal of a methodology, which starts from the raw data leads to a set of optimized prediction models. Each decision was taken, this methodology is made, taking into account a trade off between the quality of the result and the computation time needed. The methodology is composed of 5 consecutive steps:

1. Initial evaluation: Perform an initial training used as an exploration of the data and as a base to compare the next steps
2. Windowing optimization: Calculate the windowing schema optimum for each event
3. Model selection: Compare the performance of several algorithms to create the models and select the appropriate one for each event
4. Feature selection: Reduce the set of events which influences each of the events
5. Sampling: Search the best sampling technique to deal with an imbalanced dataset for each event

Chapter 6

Design and implementation of a discrete network event simulator

In this chapter, I am going to introduce the design of a data generator capable of generating a dataset of the same characteristics as described in Chapter 4. This synthetic data generator or simulator can generate real-time data, simulating the network monitor's output or generating a dataset that covers a particular time range. The core of this design consists of two similar pieces: the prediction models obtained after applying the methodology of Chapter 5 and statistical models that reproduce the times between observed events on the stage. Therefore, the system will generate new synthetic event data using those that can provide prediction models and not statistical models.

The design of this system also meets several requirements, as specified in Chapter 3. The first is the scalability of the system. My scenario is a medium-sized network with only 22 devices, but this system should be able to be applied to an arbitrarily larger scenario. To achieve this, all decisions that are made aim to create a distributed and highly parallel system, specifically through a network of software agents. In general, each agent will be responsible for one of the models, whether prediction or statistical, in isolation and independently. In addition to model-agents, another set of agents handles other tasks such as event queue, system output, or synchronization.

6.1 Design

The architecture I propose is based on a software agent system. These agents are separate processes with their own assigned resources on which a message-based

communication interface is implemented. Through this message-based communication, agents can send results or make requests to other agents to build a complex system using simple parts. In this section, I describe the architecture focusing on the different types of agents that compose it and the established communication between them. To test the feasibility of the solution empirically, I have implemented a prototype following this proposal in Scala language, making use of the Akka library for agent design and two possible options for handling machine learning models: Apache Spark for the assumption of working in a distributed environment and a large scenario, and SMILE [113] for execution on a local system and a more modest scenario size. The language features and libraries are described in Appendix A. None of the design proposal techniques is specified in any programming language or library, so this simulator can be implemented using other programming languages or equivalent libraries. In my case, I chose Scala because it allows to design parallel and distributed systems in a simple way, Akka, because it provides a simple but powerful agent system, and Apache Spark or SMILE because they have implementations of the most common machine learning algorithms.

Broadly speaking, the simulator I propose can be understood as a real-time prediction system, such as the one described in [9], which is feedbacked with its own predictions. Predictions are added to an event queue, which simulates these events' generation as if it were the output stream of the network monitoring system. Not all events in my scenario can be modeled for prediction, so prediction models will be accompanied by statistical models that simulate non-modelable events' behavior.

6.1.1 Agent based architecture

From an overview, the system consists of three types of agents. The first is the agent that handles the event queue. This agent pulls events out of the queue at their corresponding time and sends a message to other agents reporting the generated event. The second type of agent is the one that handles the output of the system. This agent's job is to apply the specified formatting to events generated by the event queue and write the system output by the decided channel. Finally, the third type of agent will be the models for each of the events. These agents will listen for all messages sent from the event queue and, based on internal logic, and independent of the other agents, will decide when the next event should occur. When this decision is made, the corresponding agent sends a message to the event queue to cause the corresponding event generation in the decided time. This last type of agent uses either a prediction model or a statistical model as its internal logic.

In addition to the agents, there is a process that is responsible for setting up the system and another component that allows training the models used by the agents. Figure 6.1 shows an architecture outline.

Figure 6.1: Simulator architecture

6.1.2 Agent communication

Communication between different agents is based primarily on two types of messages: generating an event and adding an event to the queue. The event generation message is always sent from the event queue agent, and the recipients are the rest of the system agents. This message must be answered with an acknowledgment from each of the model agents. This acknowledgment is sent when the model agent has finished recording the event's occurrence in its individual memory. Upon receipt of all acknowledgments, the event queue agent knows that the system is ready to continue the simulation and will generate the next event. The message to add an event to the queue has two fields: the instant time the event should be raised, and the list of event indexes generated instantly.

Figure 6.2: Agent communication

The message to add an event to the queue is generated on a model agent, and the recipient is always the event queue agent. This message is generated when the model agent decides that the event it handles should appear. This process is detailed later in this chapter. The message has two fields similar to the previous one: the first is when the model agent decides that the event should be raised, and the second is the index of the event handled by the agent.

Acknowledgments are only generated in the cited case because it is the process in which strict synchronization is required. In all other cases, messages are sent, and it is not needed to wait for confirmation by the recipient to continue working.

6.1.3 Event queue agent

This agent is the core of the simulation. From a data structure point of view, the agent will handle a priority queue in which tuples are stored. These tuples have two fields, an instant of time, and the event's index to be generated at that instant. The queue's priority is decided based on the time field, with the next event to be generated at the beginning of the queue.

There can be multiple records in the queue that have an equal time value. Therefore, when the agent decides to raise an event, the first item is taken out of the queue, and all those that follow it with the same time value. Then, all extracted event indexes are grouped, and a single event generation message is created, as explained in the previous section.

This agent's behavior depends on the mode of operation of the simulator: real-time simulation or batch simulation. In both cases, the system uses a time scale in which the start of the simulation is always instant 0, and the value of the time variables is the number of seconds that elapse from the initial simulation instant. For example, an event that is raised within five minutes of system startup will be generated at simulation time 300, and then the output manager agent will transform this value into a date format.

When the system operates in real time, the agent is activated periodically every second. When activated, it checks the value of the first record in the event queue. If the time value is equal to or less than the current simulation time, the agent proceeds to raise the event. In the other case, the agent does not do anything. This activation is repeated indefinitely until the program stops.

When the mode of operation is the batch generation, the system should operate as fast as possible. In this case, the agent is always active and repeats in loop raise events and wait for acknowledgments. Thus, as soon as the model agents have finished processing an event generation message, the event queue agent directly generates the next one. This loop ends when the last event generation message has a time that corresponds to a value equal to or greater than that specified in the configuration. For example, if it is specified that the simulation should cover a period of 100 minutes, the agent will stop when the last generated message has a time equal to or greater than 6000.

6.1.4 Output manager agent

This is the most straightforward agent of all. Its function is to simply write the generated events to the specified channel and in the appropriate format. The default output channel is the standard output, but it can be configured to write to a text file, an Apache Kafka channel, or other systems. When the simulation begins, this agent saves the start date set, corresponding to the zero simulation time. In this way, it will calculate the instant of time or date corresponds to each of the generated events. For example, if the start date is 01/02/2020 17:00:00, an event that is raised at simulation time 300 corresponds to the date 01/02/2020 17:05:00.

The agent will write one line for each event that simulates, even if there are several simultaneously. In my case, the event index is all the information I need, so just the date and index of the event are written, but a format map can be configured to make the output look specific. E.g., include the event error code, the machine on which it happens, or its severity.

6.1.5 Model agent

Each of these agents will be associated with one of the events in the scenario and decide when this event should be raised. To make this decision, a model that can be of two types is used. The first case is that it is a prediction model like those seen in the previous chapter. The second case will be a statistical model that just replicates the time distribution of the associated event. I will call regression models to this second type of model because their output will be the time that must elapse until the event's next occurrence.

These agents receive event generation messages and use them to monitor the system and know the current simulation time. Each of the agents can monitor the system. Differently, some of them will be fixed in some events and others in different ones. This will be decided with the configuration at the start of the simulation.

The simulation configuration file must contain the specific settings for each of the events. For regression models, the configuration must indicate the event's index and the path in which the model is stored. For prediction models, the observation window must also be specified, the indexes of the events that are taken into account in that observation window, and the lead time.

When one of these agents, based on the event generation messages received by the agent and the output of its model, decides that the event it handles should be added to the simulation queue, it will generate an add event message to the queue. The index of the event included in the message will always be that of the agent handled. The time depends on the type of model the agent uses. In the case of a prediction model, the time will be equal to the current simulation time plus the lead time. For regression models, the time is equal to the current simulation time plus the model's output.

6.2 Simulation models

The generation of each or not of the events during the simulation is decided by a model that is associated with that particular event. Each time an event is raised from the event queue, the agents that handle the models receive a message with the generated event information. This way, agents monitor the system and create particularized observation windows for each event or simply calculate time differences. The models used in the simulator fall into two categories: prediction models and regression models.

6.2.1 Prediction models

The first set of models are those obtained by applying the methodology of Chapter 5 to the data presented in `prettyrefchap:scenario`. These models start from the general idea that observing system behavior over a given time can be found in event occurrence patterns that will be used to predict future events. As described in Chapter 5, these models are generated with standard Machine Learning algorithms and specifically binary classification models: that is, the output is binary, or the model predicts that the modeled event will happen or predicts that it will not.

From the original dataset, I only study those events with at least 10 samples, a total of 168. Not all these events can be modeled. In other words, there are no event patterns that precede their occurrence. Specifically, of the 168, 99 events can be used to generate a minimally effective model. Nevertheless, not all events in this subset can generate a model of the necessary quality. This "required quality" is a threshold that must be defined according to the project's needs, in my case, as in other decisions made in this work trying to be guided by the data. Figure 6.3 shows the models' evaluations for each of the 99 events after applying the methodology of the previous chapter.

Figure 6.3: Prediction models F-score

In the case of my scenario, the mean and median F-Score of the models are similar. So I set the threshold to 0.4, which means staying with the top two quartiles' models, knowing that all of them have at least one F-Score equal to the average of

all obtained. After applying this filter, the number of modeled events is reduced to 49, and their F-Score distribution is represented in Figure 6.4.

Figure 6.4: Prediction models F-score filtered

Having selected the events to be simulated with prediction models, I save each of them' configuration and a version of the filtered, labeled dataset and with the corresponding sampling algorithm applied. In this way, an event's configuration consists of the algorithm that generates the model and its parameters, window schema, selected features, and training dataset. Using this configuration, a system module, described below in this chapter, is responsible for training the models and storing.

During simulator startup, when an associated agent is created, this agent receives as configuration: the event index, the duration of the observation window, the lead time, the selection of model features, and the location of the model. During the simulation operation, the agent follows each of these agents following the workflow shown in the diagram in Figure 6.5.

Figure 6.5: Prediction models agent workflow

The agent is activated each time it receives an event generation message. First, it checks the events generated and applies the transformation of the event indexes based on the features selected of its model. It then updates the observation window. If necessary and discarding, adding the new events should no longer be in the observation window, taking into account the event's time. If the vector representing the observation window has changed since the last iteration, the vector is passed as input to the model and makes a prediction. If the prediction is negative, nothing happens, and the agent waits for the next message. If the prediction is positive, a message to add an event to the queue is created; the event rate in the message is the one handled by the agent, and the time is equal to the time of the message being processed plus the lead time. The event will be added to the queue in the future at a time distance equal to the lead time. After sending the message to the event queue agent, the agent waits for the next event generation message. Whether the prediction is positive or negative, the agent sends the corresponding ACK to the last message before waiting for the next message.

6.2.2 Regression models

The second type of models that make up the simulator are models whose output is the time that elapses until the next occurrence of an event. As in previous models, each model handles a single event. According to the usual nomenclature, these models could be classified as regression models, as their objective is to calculate a magnitude, in this particular case, an amount of time. However, studying how to model my data scenario using regression algorithms is as big a task as doing it with classification models. Instead, in order not to deviate from the contribution proposed in this chapter, I decided to use more straightforward modeling techniques that still provide me with an acceptable result. These models will be purely statistical and try to replicate the time distribution between events. In other words, the model's output is the time that will elapse until the next occurrence of the modeled event.

For each event I decided to model, I will calculate a new variable incorporated into each of its samples. This variable is calculated as the difference in seconds with the next sample of the same event: the time difference between occurrences of the same event. The purpose of the models will be to generate events that follow the same time distribution as this new variable's values.

Figure 6.6: Time differences between event occurrences

Figure 6.6 shows the histogram of the time difference between the occurrences of an event for different events. In all subfigures, the distribution of these times does not follow any usual statistical distribution, but several of them overlap. In Figure 6.7c, for example, the distribution consists mainly of two peaks, one close to the coordinate origin and one around the 24 hours. Analyzing these peaks separately, I may be able to obtain a parameterized statistical model that is effectively adapted. However, this is not a process that can be automated, making it impossible to apply it in my scenario as I would have to apply the process manually to each event separately. Instead, some techniques allow obtaining a curve that approximates the distribution [114]. Specifically, I have used Kernel Density Estimation (KDE), detailed in the appendices. This technique constructs the approximation, assuming that each of the data samples can be modeled with a Gaussian distribution centered on the sample (in practice, this distribution can be adjusted by varying parameters such as variance or the distribution model itself). Overlaying all these individual distributions generates the curve that represents the estimated distribution. Figure 6.7 shows the distributions previously seen along with the estimate generated by this technique.

Figure 6.7: Time differences between event occurrences with KDE

Instead of a Gaussian distribution, other distributions such as the top-hat can be used. Other distributions may have the advantage of being more accurate in the estimation, but in the case of a simulation, the inaccuracy can become a bene-

fit. The data that is generated in a simulation must have the same characteristics and behavior as the original scenario, but they must not be exactly the same. The only consideration I should have with Gaussian distributions is those events with minimal temporal differences. In these cases, the distribution queue can span negative values, as shown in Figure 6.7. When representing times until the next occurrence of the event, it makes no sense to contemplate negative values; and even because two equal events are not considered simultaneously, the value 0 also does not make sense. To solve this, I simply truncate the estimated distribution obtained, keeping only values greater than or equal to one.

These distributions can capture and reproduce repetitive behaviors (e.g., an event that occurs every 24 hours). However, they cannot record that an event only has an activity for a limited period within the dataset's total time. For example, suppose there was an event that happens only in January repeating each day. In that case, the model will generate a sequence with the same characteristics with which the event was initially repeated, but throughout all the simulation time, because the information that the event only occurs for one month is not recorded in the time difference variable between events. I have a limited amount of data in my scenario, and I cannot assume assumptions such as that an event only occurs for a few specific months, since I do not have enough data to corroborate the hypothesis. To mitigate this effect, the solution I have used is to add two synthetic samples for each event, one at the beginning of the dataset and one at the end. This way, the distribution can consider the time between the start or end of the dataset and the event's activity period.

This approximate distribution with KDE has the characteristics of a probability distribution. Its integral is equal to one. This allows me to get its CDF, which will have an X-range from the minimum time difference between events to the maximum and a Y-range between 0 and 1. By swapping the axes with the curve of a function that takes values between 0 and 1, I can feed with a uniform random number generator and generate a sequence of values that follows the time difference distribution between samples for the modeled event. Figure 6.8 shows the CDFs of the distributions shown above.

Figure 6.8: CDF of time differences for sample events

Of the 168 events with at least 10 dataset samples, 49 of them are modeled using the prediction models obtained using the previous chapter’s methodology. The rest, 119, are modeled as described in this section. Unlike prediction models, these models do not have any input type, and their output is based on a random number, so when they work inside the simulator, they do not need to store any kind of record of what happens in the system.

During the simulator boot, when an agent associated with a regression model is created, it needs a configuration just like in the case of a prediction model, but in this case, simpler as only the index of the modeled event and the location of the model is required. Since there is no observation window, the agent workflow is also different, as seen in the diagram in Figure 6.9.

Figure 6.9: Regression models agent workflow

When one of these agents decides to send a message to add an event to the queue, it stores the time at which the event should be raised. After this message is sent, the agent ignores all messages until one whose time is equal to or greater than the time it has stored is received. In that case, the agent pulls a random sample from the model to decide when the next event should be generated and generates the message to add an event to the queue. This way, there will never be two records of the same event in the event queue of those simulated with this type of model. With these models, the next occurrence of an event is calculated after the previous event has occurred. Finally, as with prediction models, before waiting for the next message, the agent sends the ACK of the last message.

6.3 Simulation workflow

6.3.1 Startup and set up

In any of the simulator operating modes, it is necessary to provide a system boot sequence. This sequence can be a fragment of the original dataset or a specifically generated sequence. In addition to the boot sequence, the seed to be used for random number generators used in the system can be specified in the configuration.

The boot sequence is added to the event queue so that the simulator's initial output will be this same sequence of events. These events are not added to the queue using agent-to-agent messages, but the event queue agent adds them directly from the startup configuration file.

6.3.2 Model trainer

The simulation uses pre-trained models, so I include an option that allows to generate these models and save them to a specific location, starting from a configuration file that specifies how the model should be generated for each event. Model training depends on the type of model, in the case of regression models, there is no training as such. The configuration file must specify the points of the curve that represents the model.

When training and storing models, this process also generates the configuration file that will be used during simulation startup and contains the parameters described in the previous sections.

Models are trained using Apache Spark or SMILE. The generated models are not supported between both libraries, so the simulation must be run using the same library that was used to train the models. In the case of my application scenario, SMILE is sufficient for the volume of data being handled, so it is the option I have used in my experiments. For a larger scenario, with more events or more complex models, Apache Spark can be used.

6.3.3 Messages

Messages are designed to contain the minimum information required, which in most situations is an event and associated simulation time, or a list of these. The exception to this is the agent engine's own messages for the actors' spawn and the messages that the models send when they are created to the event queue agent to register.

All message sending and receiving are asynchronous, the only sync mechanism is the one described for batch generation mode.

6.3.4 Simulation end

There is nothing particular about the completion process of the simulation. In the case of a batch simulation, when the target time is reached, the event queue agent invokes the supervisory process that stops all agents and terminates the process.

In the case of real-time execution, the system only stops with an external halt signal.

6.3.5 Working modes

Real-time mode

The real-time operating mode simulates the output of the network monitoring system. Each second of real time corresponds to one second of simulation, so all calculations required to process an event generation message must be executed in less than a second to prevent congestion. In this mode, the agent does not limit its rate expecting all ACKs from the previous message to be received, but that a scheduler activates the agent every second regardless of whether all of them have been received. For regression models, the agent can be in two situations when receiving a message: having to calculate the next occurrence of the event or continuing to wait for the simulation to arrive at the time that was calculated the previous time. In either case, the processing time is minimal. Figure 6.10 shows the time distribution it takes for an agent to send an ACK taking into account the two possible situations.

Figure 6.10: Regression model agent response time

Something similar happens in prediction models, in this case, the agent can take three paths: if updating the observation window, it does not change, the ACK is sent immediately if there is any change a new prediction is made that can be positive or negative. Only in the case of making a positive prediction is the message of adding an event to the queue sent. Therefore, there are three tasks that

are performed and from which I can measure their duration: get the new observation window, run the model, and send the add event message to the queue. As for regression models, Figure 6.11 shows the observed processing times for prediction models based on the path taken.

(a) Prediction positive

(b) Prediction negative

Figure 6.11: Prediction model agent response time

In this case, the last peak observed in the graph is less than one millisecond. This is the maximum time that an agent with a prediction model takes to do its process, and the value is much smaller than a second. However, in every second, all models must complete their execution. Figure 6.12 shows the time distribution

observed in running all models, that is, the time that elapses from when an event is pulled from the queue until all acknowledgments for that message are received.

Figure 6.12: General response time

Again, the distribution's peak is at a value much less than one second, approximately in 5 microseconds. This measurement is obtained with the models generated for my scenario and running the simulator on a laptop, Section 3.3.2. Therefore, the growth margin to adapt the simulator to a broader scenario is wide, increasing it even more by running the simulator in a more robust environment.

Batch mode

Operating in batch mode, there is no time limitation to be met. However, to ensure proper system operation, the previously named synchronization method must be incorporated to ensure that all models have finished processing the receipt of a generated event before proceeding with the next one. In this case, events are not generated following a simulation time equal to real time, but the next event in the queue is always raised regardless of the difference in simulation time with the previous one. Therefore, the evaluation of the simulator's operation in this mode must be measured differently from that of real time. The metric I have chosen to evaluate is the runtime required to generate a dataset covering the same range as the data in my scenario, 206 days. Besides, I have repeated this process by extending the target period to one year, five years, and ten years. The results are presented in Table 6.1 on page 103. These results are calculated as the average

value of 10 test runs. The results prove that the simulator can be used to generate large datasets in a short time, around a year of data per minute of simulation.

Target	Duration (s)
206 days	55
1 year	66
5 years	269
10 years	409

Table 6.1: Simulation duration in batch mode

Another way to evaluate the simulator in this mode of operation is to calculate the ratio of events generated per second of execution. By calculating this metric again as the average of 10 test rounds, I have obtained that the ratio is 650 events for every second of execution.

As in the previous case, the measures cited in this section have been generated for the models in my scenario. Again, the margin for incorporating more complex models or scenarios is wide.

6.4 Chapter summary

In this chapter, I have described in detail my proposal for an event simulator based on prediction models and regression models. The system is based on an agent architecture that works around a priority queue in which events that are predicted are added, and those events are taken out when they are simulated. One of the agents handles this event queue. Another agent works with the output of the simulation. Furthermore, the rest of the agents are each associated with a system event, and their function is to decide when to simulate the occurrence of that event based on a model.

I have then described the two types of agents that I use to generate simulated events: regression and prediction. Regression models are generated by estimating the time distribution of each event using KDE. Prediction models are trained with the configuration obtained from the methodology proposed in the previous chapter.

The simulation system has two operating modes. Real-time operation simulates the monitoring system's output, and the simulation time is equal to real time. The batch simulation mode generates the system output for a specified period of time,

running at the highest possible speed. According to the experiments conducted in this mode, 650 events are simulated for every second of the simulation run, so to generate a dataset covering the same amount of time as my scenario, the simulator takes 55 seconds.

Chapter 7

Validation

This chapter explains how I validated both the models obtained with the methodology proposed in Chapter 5 and the simulator presented in Chapter 6. Although they are different contributions, some of the results of this chapter are closely related. And so is why I separated the validation of each proposal from their respective chapter, and I include both together in this chapter.

The first part of this chapter is dedicated to validating the models obtained in Chapter 5, not only as classification models but also as an element of an Online Failure Prediction system with all its implications.

The second section is the validation of the simulator from Chapter 6. First, it is validated as a software module that should satisfy the requirements proposed in Chapter 3. Then the data generated with this module is compared to the original data analyzed in Chapter 4.

7.1 Prediction models validation

The evaluation of the prediction models I make in Chapter 5 is done from a machine learning approach. Each of these models is evaluated using the F-Score metric and a K-Folds Cross Validation. The goal of these models is to act on a real-time prediction system, an OFP. Therefore, to validate that the models are suitable, it is not enough for me to evaluate them from the point of view of machine learning, but I must also evaluate their performance as parts of a prediction system.

The F-Score metric is obtained from the confusion matrix, in which the model response against the actual value for a series of samples is evaluated. These sam-

ples are temporary captures of the monitored system state. As I define the window schema, a model receives as input a vector representing the observation window, and a positive prediction is only considered correct if the target event occurs exactly over the course of a time equal to the model's lead time. If the event occurs, even if it is only a second later than the lead time, the model response is considered incorrect from the point of view of machine learning evaluation. However, if I consider that this model works in a real-time system and generates a prediction every second, this situation should be interpreted as a success. In addition, observation windows are usually long, and events that trigger positive prediction in the model will be reflected in the input vector until they exit the observation window. This implies that the model will make positive predictions as long as the observation window is maintained, but the target event will only occur in an instant. Therefore, when evaluating models as part of an online prediction system, I interpret positive predictions as warning signs that the target event will happen in the time that they remain active. To numerically evaluate these alerts, I define the concept of Alert Session.

An alert session is defined as the period of time between a positive prediction of a model and the next negative prediction for that model. That is, the time that the alert is kept active. Taking into account this concept of an alert session and whether or not the target event occurs, three different situations can happen:

1. The target event occurs during an alert session: this is the situation where the model is considered to have behaved correctly.
2. The target event does not occur during an alert session: this is a false positive situation. The model has alerted that the occurrence of an event that finally has not happened.
3. The target event occurs outside an alert session: in this situation, the event has occurred without the model alerting.

These situations can be equated to different values within the positive true confusion matrix, false positive and false negative, respectively. However, I do not contemplate the concept of true negative as it is not interesting from the point of view of prediction.

The duration of alert sessions is closely linked to the model's length of the observation window. The longer the observation window spans, the more likely it is that the events that trigger the positive prediction will be in the observation window for longer and, therefore, longer alert sessions. This is one reason why in step 2 of the methodology I described in Chapter 5, the goal I set when optimizing

the window scheme was to maximize the model score, minimizing the observation window.

The algorithms I have used to generate the models have random components in their training. However, its output is deterministic. For the same input, it will always give the same output. Models have no memory or random components when calculating their responses. Taking this into account, I can analyze the behavior of the models and alarm sessions that generate during the monitoring period of my dataset, evaluating the model's response only when its input changes. That is, every time an event enters or leaves its observation window. For each of the events that affect model input, I evaluate the model response when the event is in four moments of time relative to the observation window: the moment before the event enters the observation window, the instant the event enters the observation window, the last moment the event remains in the observation window, and the moment it is no longer in the observation window. Thus, in all possible transitions in the model output, I have the value of the previous and subsequent output. With these values, I can determine what the output of the model will be at any instant of the monitoring period. By grouping the intervals, I get the alarm sessions generated by the model throughout the monitoring period.

The first result I study in this analysis of alert sessions is the number of sessions each model generates based on the number of samples in the target event. In Figure 7.1, I represent this comparison. There is one point for each event, the x coordinate is the event's frequency, and the y coordinate are the number of sessions generated by the model. There is also a blue line, which is the regression trend of the points plotted, and the red one is the line with slope equal to one, just for reference. The dispersion is not very high, but some model events with few samples generate too many alert sessions, which involves many false positives. On the other hand, event models with many samples generate fewer alert sessions than samples. This does not reflect poor model behavior because multiple occurrences of the target event can occur in an alert session, and therefore events that occur in bursts will typically have a model that generates fewer sessions.

Figure 7.1: Number of sessions versus event frequency

In the following Figure 7.2, I have drawn with stacked bars, for each event, the number of true positives and the number of false negatives. The sum of these two quantities is equal to the number of samples of the event. The ratio between the number of true positives and the bar's total height is equal to the recall: the number of correct predictions over the number of positive samples. Some of the models are not able to predict any samples and others are able to predict all. The mean of the recall is 0.35, and the median is 0.25. In total, of the 11485 events in the dataset, the prediction models were able to alert of 4875, 42% of the total.

Figure 7.2: Number of True Positives and False Negative sessions of each event

Finally, considering the false positives, there is also no obvious relationship between the number of False Positives and the event's frequency, or the recall obtained. The metric that I have used and many of the techniques applied do not penalize much the appearance of false positives. Throughout this work, I have preferred to increase the number of true positives, even at the cost of also increasing false positives. However, not in all scenarios is it possible to make this decision, in which case this effect should be considered from the processing of the data at the beginning of the process. In figure Figure 7.3, I added a bar with the number of false positives for each event. For most events, the number of false positives versus the number of true positives is not very high; however, some events like those I point out in the image have a very high ratio. The models of these events have learned a pattern that clearly does not help predict the target event.

Figure 7.3: Number of True Positives, False Negative and False Positive sessions of each event

7.2 Simulator validation

In the previous chapter, I presented the results of the experiments I used to validate the simulator from a scalability and performance point of view. In this section, I will describe the validation I have performed of the data generated by the simulator; this is the output of the system. In these tests, I have used only the batch mode of operation since most of the checks that I will perform will be comparisons with the original dataset. I have configured the simulator to generate a synthetic dataset that spans the same period as the actual set, i.e., 206 days starting with 2014-09-16 03:41:49.

Representing the number of daily events of the generated synthetic set, I have observed that the obtained series needs a stabilization time until the simulator reaches a stable output. This is because at the beginning of the simulation, all regression models make their first estimate of the occurrence of the next event, and it is usually the case that these first events are grouped in the first moments of simulation. To solve this, I just start the simulation on a date earlier than the desired one and discard the first results. In Figure 7.4 I plotted the graph of the number of daily events in 10 different runs of the simulator. The dashed line marks the time instant 2014-09-16 03:41:49, which is the date of the first sample in the dataset; events prior to this date are discarded. All curves have much higher values to the left of this division, which on the right once the simulation is stable. Based on these results, it follows that the simulation takes about 9 days of simulated time to stabilize. To ensure a margin of safety, I have set this discard offset of the first data in 15 days of simulation time (7% of the dataset extension).

Figure 7.4: Simulation output stabilization

To check the similarities and differences between the synthetic dataset and the original dataset, I start by checking the more general aspects of both. The first aspect is trivial, but it is necessary to be checked and is for the two datasets to span the same time period. As the simulator is designed, which generates its output indefinitely until it reaches the target date, this is a fact. The next thing to validate is the number of records and their distribution. From this point, the synthetic dataset already has a big difference from the original, not all events that appeared in the original set can be modeled, so not all events will appear in the synthetic dataset. I have discussed this limitation in previous chapters; only events with at least 10 samples are modeled, either with a prediction model or regression

models in the simulator. To make a fair comparison between the two datasets, I discard all events with fewer than 10 samples from the original dataset. Having applied this filter, the original dataset has 13996 records, and the synthetic dataset has an average of 11369 records (extracted from 10 runs). This represents 20% fewer records, but if the peak of December 24 (Chapter 4) is not counted, the difference is only 10%. In general, the synthetic dataset behaves according to the typical distribution of what is observed in the scenario, and extreme situations will not be represented. Figure 7.5 shows the number of daily events in the original dataset and one of the synthetic runs.

Figure 7.5: Daily events for original and synthetic datasets

Then I measure the number of times each event appears. Figure 7.6 shows the event occurrence distribution in the original dataset and the same for the synthetic dataset in two subfigures. I have colored the bars that correspond to the events that are modeled with regression models blue, and the prediction models in red. The big difference between the two figures corresponds to the prediction models. In some cases, there are too few samples or none (e.g., the event 0 does not appear on the simulated dataset). In other cases, there are too many, occasioned by models that trigger their prediction with too general patterns. The main reason for this is that there are few prediction models compared to regression models and the former are those which can replicate the relationships between events. In a scenario where more prediction models could be achieved, this difference would be much smaller.

(a) Original data

(b) Synthetic data

Figure 7.6: Event counts for original and synthetic datasets

By performing these same checks using a simulator configuration in which all events are modeled with regression models, the results change significantly. The total number of records rises to 14228, surpassing the original dataset. The apparition distribution of each event is practically the same as in the case of the original scenario, as it can be seen in Figure 7.7. This figure shows the event distribution of the original data and the synthetic data overlapped.

Figure 7.7: Event counts for original and regression only synthetic datasets

On this occasion, the synthetic data set has a very similar event distribution to the original dataset. The mean absolute error (MAE) in the number of samples for each event is 21.79, while in the previous case, it was 32.45. In Figure 7.8, I have re-drawn the time distribution of the number of daily events compared to the new simulated set. In this case, the MAE is 39.48, while in the data seen in Figure 7.5, it was 44.21.

Figure 7.8: Daily events for original and regression only synthetic datasets

It is not necessary to check the temporal difference between occurrences of equal events since this is the basis of the model with which events are generated.

However, although, statistically datasets are very similar, when using models that only take into account the event they model the same, the relationships between events disappear. This is the reason, as I previously named, for the poor functioning of prediction models in simulation

7.3 Chapter summary

In this chapter, I presented validation of the prediction models obtained in Chapter 5 and the datasets that can be obtained with the Chapter 6.

To validate the prediction models, I have defined the concept of alarm session. This concept refers to the period of time in which a prediction model makes a positive prediction continuously. Using this concept, I have evaluated the operation of the models with the dataset of my scenario, emulating the input of an OFP and evaluating how many alarm sessions captured the target events and how many events were unpredicted.

Validation of simulated datasets has been performed against the original set. I have verified that the temporary event distribution matches that of the original set. However, events generated with prediction models have extreme behaviors. They generate too many samples or too few. By repeating validation against a simulated dataset only with regression models, I have verified that the result is statistically more similar to the original dataset. I have also concluded that simulation with prediction models works best, the more prediction models in the simulator, as the better the event patterns needed to trigger predictions will be better represented.

Chapter 8

Conclusions

In this last chapter, I present the conclusions of my research. I start by explaining if my proposed solutions achieved the objectives set in Chapter 3 with all their requirements. Then I discuss the strengths and weaknesses of my solutions. Finally, I propose some future lines to continue with the research presented in this document.

8.1 Objectives

My first objective was to propose a general methodology to create failure prediction models based on machine learning algorithms. This objective is addressed in Chapter 5. The methodology consists of 5 steps based on the preprocessing and evaluation stages of a prediction model. The first step is to perform an initial evaluation with basic models used as a comparison point. The second step is to optimize the observation window, in my case, through a genetic algorithm. The third step is to perform a benchmark of modeling algorithms and tune the selected algorithms' parameters. The fourth step is the dimensionality reduction based on feature selection. Finally, the fifth step is the application of sampling algorithms to address the problem of data imbalance.

This methodology's steps have been defined generically and decisions made are based solely on the nature of data used. Therefore, the methodology can be applied to new datasets, techniques, or algorithms without significant changes. Decisions made do not require human interaction except in the configuration and choice of metrics and algorithms used. The entire methodology has been implemented and evaluated in Python with scikit-learn, making the solutions independent of the execution platform. The generated code is published in a public repository. The solution has also been tested with Apache Spark allowing it to be applied to larger

or more complex scenarios. The execution environment used is documented, as well as the times used to run each phase. In each of the stages, I have detailed the decisions made to maintain the commitment between time and results. The evaluation of the generated models is based on a selected metric after exploring possible alternatives, is possible to modify it if required.

My second goal was to design a simulator using the prediction models obtained in the methodology. This contribution is described in Chapter 6. The simulator I propose has an agent-based architecture, in which each agent simulates occurrences of a particular event. These occurrences are decided using prediction models along with regression models for events from which a viable prediction model could not be obtained. These regression models are generated by replicating the distribution of time differences between events with the KDE technique. The simulator has two operating modes: real time and batch generation. The proposed architecture has been tested using Akka in Scala. The models have been tested with the SMILE library and Apache Spark..

The source code for this solution is also published to my public GitHub repository. The proposal has been implemented with Akka, but it can be translated into other agent systems without the need for design changes. The simulator can run on a modest machine such as a laptop or scale to a distributed architecture to simulate more complex scenarios.

I consider that the proposed objectives are resolved in the two contribution chapters, and the proposed solutions are validated in Chapter 7. From these proposals, I have been able to obtain some conclusions and analyze their limitations, as I describe in the next section. All the artifacts described in this document are published under an MIT license in my GitHub repository <https://github.com/jandion/>. The dataset used has been anonymized, removing all confidential variables, and published on IEEE Dataport [115].

8.2 Discussion

The performed experiments use a data set obtained from real industrial infrastructure. This fact gives empirical validity to the results. However, using this dataset also carries some risks. The first is the insufficient number of samples for some events. Using a single dataset can then generate skewed responses. I have tried to avoid the effects of these two facts by performing evaluations with cross validation and trying to keep agnostic to the data I worked with, that is, decisions are made based on the data, not the knowledge of the domain.

In Chapter 4, I discussed the need to filter out less frequent events. I set a minimum threshold in 10 samples, discarding all events that did not reach that amount. Only selected events are used to both generate prediction models and in the simulator. However, all events, including those that did not reach the sample threshold, were used to train prediction models. I made this decision because I did not want to discard the information that these rare events could provide, no matter how minimal it was. In hindsight, I do not think it is worth saving these events for training. If used in training, the models' input vector is much larger because it must have as many positions as events to evaluate. This generates models that, in many cases, are unnecessarily complex. The feature selection step instead resolves this fact by reducing the input vector and considering only influential events in each model. However, this process is much more expensive due to the high number of events. The methodology's previous steps can also be executed more quickly if these events are not taken into account from the beginning. Therefore, I think it would have been useful in my scenario to discard events that did not exceed the filter also from the training data. Nevertheless, it is a risky decision as useful information for predicting some models could be lost.

The simulator's architecture is designed to escalate and adapt to more significant scenarios. The actors' models can be replaced by any kind of data generator to adapt the simulator to new situations. The agent intercommunication is simple, but it also can be easily expanded, creating a communication protocol between agents for coordination or similar tasks. The data collected from my scenario was enough to create several prediction models that can be successfully used in an online predictor system. However, as I described in the validation chapter, this number is too low to replicate the complex interactions between events so that the simulator output may lose these relationships. However, the output is still useful as it has the same statistical properties as the original data. The real time operation can be used to test systems like an OFP.

Regarding the reproducibility of this work, it has been one crucial aspect of my research as I propose a methodology that can be used by other researchers than my team. I have detailed all the decisions made and have published all the generated artifacts in my GitHub repository as well as the anonymized data in IEEE Dataport. All the intermediate results are included in Appendix C. Anyone interested can replicate the experiments and analyzes that I have presented in this document using these resources. The time required to run these analyzes can also be approximated by comparing the hardware and runtime environments that I have used with those that will be used. The same reasoning can be applied to

replicating these experiments in a new environment, a different domain, or a new data set..

8.3 Future work

In this section, I list some of the possible paths to continue with this research line. Some of them have been discovered during the research process, and others are based on my application scenario's closed scope.

Incorporate new algorithms

The first branch by which it is possible to continue with the work presented incorporates new techniques and algorithms into the methodology. The methodology is designed to work with an arbitrary set of methods. For example, in the case of algorithms, I have used a selection based on those available in the Spark Mllib module. This set of algorithms is a good representation of classification algorithms, but it is possible to incorporate new algorithms into the list. The same goes for sampling methods to deal with data imbalance or the window size optimization algorithm. All these points are variables that can be modified.

New datasets

I have done my research and validated my solutions with real data but obtained only from one scenario. Both the methodology and the simulator can be tested with new scenarios from different domains.

Alternative dimensionality reduction

The variable reduction methodology step has been carried out by removing iterative features based on the importance of these. There are several ways to perform dimensionality reduction, from applying aggregation functions to other methods of evaluating the importance of features.

Increase simulator models' complexity

Simulator's regression models are based only on the time differences of one event with itself. These models can be more complex, for example, by adding as input the time elapsed since the last event occurrence would add memory to the model instead of being invariant. Another option is to model the time to the next occurrence based on the time lapsed to the occurrence of all other events. Those models would use an input vector like the prediction models.

Appendix A

Tools and software

A.1 Programming Languages

A.1.1 Scala

Figure A.1: Scala language logo

Scala combines object-oriented and functional programming in one concise, high-level language. Scala's static types help avoid bugs in complex applications, and its JVM and JavaScript runtimes let you build high-performance systems with easy access to huge ecosystems of libraries.

Scala is a programming language designed to run on the standard Java Virtual Machine, which unifies object-oriented and functional programming [116]]. Martin Odersky started the language, and its first version was published in November 2003. It is a high level, statically typed and compiled language [117]. It is published under an Apache License 2.0, making it a free and open source.

It is designed to interoperate with Java and all its libraries fully. It shares many object-oriented features with Java, combined with functional ones like lazy evaluation, pattern matching, or immutability. Immutability, in a programming language, means that a declared variable can not change its value. This property

makes this language easy to use in concurrency: parallel or distributed computing. When it is compiled, it generates JVM (Java Virtual Machine) bytecode to be executed on any system that supports the JVM.

Scala's name comes from "scalable language," and as it can be expected, this language stands out in scalable systems such as cloud computing or distributed systems in general. Many open-source solutions for Big Data or distributes are written in Scala: Apache Spark, Apache Kafka, Apache Flink, or Akka.

This language has been used in this thesis for multiple reasons, among which the complete compatibility with solutions for distributed processing and all Java libraries stands out. Most of the tasks carried out in this work are parallelizable, something easy to achieve with this language. Being a compiled language, I can achieve greater efficiency in tasks than using interpreted languages such as Python.

A.1.2 Python

Figure A.2: Python language logo

Python is a great object-oriented, interpreted, and interactive programming language. It is often compared (favorably of course :-)) to Lisp, Tcl, Perl, Ruby, C#, Visual Basic, Visual Fox Pro, Scheme or Java... and it's much more fun.

Python combines remarkable power with very clear syntax. It has modules, classes, exceptions, very high level dynamic data types, and dynamic typing. There are interfaces to many system calls and libraries, as well as to various windowing systems. New built-in modules are easily written in C or C++ (or other languages, depending on the chosen implementation). Python is also usable as an extension language for applications written in other languages that need easy-to-use scripting or automation interfaces.

Python is an interpreted object-oriented programming language. It includes some features from functional programming. It is designed to have a clear syntax. As

it is not a compiled language, it is considered portable, and it can be run both in Windows and Unix systems. Its first version was published in 1990. The current version, at the time to write this document, is Python 3.8.5. From Python 3, some significant modifications were introduced in the language, making it not fully retro compatible.

Even if it is an interpreted language, it is efficient. Added to this simple syntax, the language has grown quickly in the last years, mostly because it has become one of the most used languages (along with R language) in Machine Learning and Data Science. Most of the current Machine Learning libraries or frameworks have interfaces to use them whit Python or implemented with it. Some of the well known examples of this are SciKit-Learn, TensorFlow, Keras, or Spark.

Python is used in this work for its versatility. It is easy to use for file management, data processing, or visualization. Used in an interactive notebook, Python is a powerful tool to perform data analysis.

A.2 Software frameworks

A.2.1 Spark

Figure A.3: Apache Spark logo

Apache Spark is a fast and general-purpose cluster computing system. It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs. It also supports a rich set of higher-level tools including Spark SQL for SQL and structured data processing, MLlib for machine learning, GraphX for graph processing, and Spark Streaming.

Apache Spark [118] is an open-source distributed general-purpose cluster-computing framework. Spark is designed to perform efficient computing in clusters efficiently. Its initial release was published in May 2014, and currently, the last stable release is the 3.0.0 from June 2020. Although the artifacts generated in this work used version 2.3.4 (a previous version from September 2019), all the code can be migrated to the latest version without any modification. Spark is implemented in Scala, making it fully compatible with Java. It also provides interfaces that use

it with Python and R Language. Apache Spark is an open source project of the Apache foundation.

Apache Spark defines a computing architecture based on a master and a set of workers. The master oversees the slice and schedule the task among the workers, and collects the final results. The workers perform the computing tasks and keep in memory the partitioned data. Spark provides a simple interface to use the clusters with implicit data distribution and fault tolerance. Spark is currently one of the most used technologies for Big Data processing and cluster computing in general.

Spark counts with several modules that enable different capabilities. With Spark SQL [119], structured data can be treated as SQL tables using queries to access or modify it. Spark Structured Streaming[120] provides an interface to consume data streams from several sources (like Apache Kafka or Apache Beam) and process the incoming data in real time. Finally, Spark ML and Spark MLlib [121] are the Machine Learning modules. These modules implement the most used data processing techniques and a wide range of algorithms for regression, classification, or clustering.

A.2.2 Pandas

Figure A.4: Pandas logo

pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language.

Pandas [122], [123] is a software package for Python used in data analytics and data manipulation. The first version was published in January 2018, and the current version is the 1.1.1 published in August 2020. The version I use in this work is the 1.0.4 from May 2020. However all the code can be run with the last version without any needed modification. Pandas is published under a BSD 3-Clause License making it open source and free.

Pandas offer an interface to handle structured data in table format, time series

data, or arbitrary matrix data. Pandas' primary data structure is the DataFrame, which contains labeled axes, rows, and columns. The columns use the secondary data structure in Pandas: Series, a one dimensional array of data. With these two data structures, combined Pandas offers a wide range of methods to process data like handling missing data, filtering, reshaping, modifying existing data, grouping data, merging and joining data sets, and input/output operations to read and write data in the storage.

A.2.3 Scikit Learn

Figure A.5: Scikit Learn logo

Scikit-learn is an open source machine learning library that supports supervised and unsupervised learning. It also provides various tools for model fitting, data preprocessing, model selection and evaluation, and many other utilities.

Scikit-learn [124] is a Python library for Machine Learning. Davi Cournapeau published its first version in June 2007. Its last version, and then used in this work, is the 0.23 from May 2020. Scikit-learn, or sklearn, is written in Python and C++. It is published under a BSD License, making it free and open source. SkLearn implements various classification, regression, and clustering algorithms covering the most used ones for each task. It also provides an excellent spectrum to work with Machine Learning models. It counts with several algorithms to perform dimensionality reduction such as Principal Component Analysis (PCA) [125], Latent Dirichlet Allocation (LDA) [126], or Feature Selection. The module for model evaluation implements classification, regression, and clustering metrics, including the most well-known scoring techniques. Also, it implements Cross Validation techniques with many of its variants like K-Folds, Stratified K-Folds, Repeated K-Folds, etc. For model hyperparameter tuning, the library implements automated Grid Search and Randomized Parameter Optimization. Finally, in the preprocessing module, there are several functions to work with both numeric or vector data, including standardization, normalization, encoding, discretization, etc.

A.2.4 Akka

Figure A.6: Akka logo

Akka is a toolkit for building highly concurrent, distributed, and resilient message-driven applications for Java and Scala

Akka is a software toolkit and runtime used to create concurrent and distributed applications on the Java Virtual Machine (JVM). It supports multiple concurrency schemas, but its principal component is the actor-based concurrency model. Erlang inspires this model. Jonas Bonér originally published Akka in July 2009. Its current version, and then used in this research, is the release 2.6.3 from January 2020. Akka is written in Scala and provides language bindings both for Scala and Java. Akka is an open source project published under an Apache License 2.0.

The actor based model uses an asynchronous message-based communication system between actors avoiding any synchronization signals. Each actor is autonomous, but they are arranged in a tree hierarchy, linking them to another actor, which acts as the supervisor. Akka provides an abstraction to host the actors through a cluster in a transparent way to the user and the message communication. The Akka engine also handles the lifecycle of the actor. The actor model provides a scalable and parallelizable way to implement high-performance solutions. The model also protects from the common problems in multithreaded applications or common pitfalls like memory sharing.

A.2.5 Other libraries and modules

A.3 Environment Tools

A.3.1 Hadoop FS

Figure A.7: Apache Hadoop logo

The ApacheTM Hadoop[®] project develops open-source software for reliable, scalable, distributed computing.

The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. Rather than rely on hardware to deliver high-availability, the library itself is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures.

The project includes these modules:

- Hadoop Common: The common utilities that support the other Hadoop modules.
- Hadoop Distributed File System (HDFSTM): A distributed file system that provides high-throughput access to application data.
- Hadoop YARN: A framework for job scheduling and cluster resource management.
- Hadoop MapReduce: A YARN-based system for parallel processing of large data sets.
- Hadoop Ozone: An object store for Hadoop.

Apache Hadoop [127] is a software library for processing large amounts of data in a distributed manner. Hadoop is written in Java and its first version was released in April 2006. Hadoop is a project of the Apache Foundation.

Hadoop consists of different modules that operate around a distributed data format. This data format is called Hadoop Distributed File System (HDFS) and is used not only by Hadoop but by many other tools in the distributed computing environment such as Spark or Kafka. HDFS allows storing data on a distributed system as a cluster, partitioning, and replicating the data across different infrastructure nodes. This partitioning allows parallel access to the data.

A.3.2 MLFlow

Figure A.8: Databricks MLFlow logo

MLflow is an open source platform to manage the ML lifecycle, including experimentation, reproducibility, deployment, and a central model registry.

MLflow is an open source platform for managing the machine learning lifecycle. Databricks E Quipo develops MLflow. The first version was released in June 2018. The project is published under an Apache 2.0 license.

MLflow's primary tool is a machine learning model tracking system. Models are filed on the platform where they are stored and organized into experiment folders. The platform allows to access registered models, compare their metrics, code, or training times. It is possible to make graphical and table-like comparisons of different models. These analyses enable the study of the influence of different hyperparameters on the results of the model.

A.3.3 Jupyter notebook

Figure A.9: Jupyter logo

The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations,

visualizations and narrative text. Uses include: data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning, and much more.

Jupyter is a code notebook platform. Code notebooks enable developing and executing code dynamically, interactively, and in an orderly manner. It is currently one of the most widely used platforms for performing data analysis, as it allows to run code in different programming languages such as Python or R and create visualizations along with the results of the analysis. The first version of Jupyter was released in 2015 and is published under a BSD license allowing free use of the platform. Initially, the project was Jupyter Notebook, but it has been expanded to other solutions such as Jupyter Lab or Jupyter Book with different features.

A.3.4 Zeppelin

Figure A.10: Apache Zeppelin

Apache Zeppelin Web-based notebook that enables data-driven, interactive data analytics and collaborative documents with SQL, Scala and more.

Like Jupyter, Apache Zeppelin is a web platform for code notebooks. Zeppelin is designed to work in big data and distributed computing environments. Zeppelin can connect to a Spark cluster to run queries and collect the results in the notebook. Zeppelin is a project of the Apache Foundation and is published as open source and free.

Zeppelin accepts Java, Scala, Python, R, and other programming languages. It is possible to create a notebook that uses different languages simultaneously, allowing to take advantage of each of them's advantages and libraries (e.g., process the data with Spark in Scala and analyze the results with Pandas in Python).

Appendix B

Machine Learning algorithms and Data Processing Techniques

This chapter briefly details and explains the concepts, algorithms, techniques, or methodologies used to develop the work presented in this document.

The first section is dedicated to Machine Learning. It details how some techniques such as K-Folds and all the algorithms used for prediction modeling are detailed. Besides, a list of hyperparameters used by each of the algorithms is included. The second section covers all the techniques related to data processing and manipulation. Moreover finally, the third section describes the basics of agent computing.

B.1 Machine Learning techniques

B.1.1 Classification models

Machine learning algorithms are usually classified by the purpose of the model they generate. The most common are regression, classification, or clustering, but there are other objective tasks such as association or dimensionality reduction. Many of the algorithms can be used interchangeably for several of these tasks, or with small adjustments. The second level of classification is related to training data: if the solution to the problem is known (e.g., tagged vectors), the problem is considered a supervised learning problem if, on the contrary, it is not known (e.g., Clustering problems) are considered unsupervised learning. Regression and classification are tasks that are most often resolved with supervised learning. In both cases, the input is a vector that can combine numeric and categorical variables. The main difference is that the output of a regression is a numeric variable, and in the case of classification, it is a categorical variable.

Classification problems are often called labeling problems. The model should learn the characteristics or patterns that are particular to each of the possible labels. In general, a model can output an undetermined number of labels or categories, but in practice, classifiers with binary output, or binary classifiers, are used, whose output is whether the input belongs to a particular category. A classifier that works with more than two categories can be built by combining several of these binary classifiers.

The prediction models that have been used in this job are binary classifiers: their output can only be whether or not the event will occur. A predictor system that incorporates several of these models can be considered a multi categorical classifier since its output will be that set of events will occur.

B.1.2 Overfitting

In some cases, Machine Learning models are over-adapted to training data by losing the generalization capacity. When this happens, it is said that the model is overfitted [128]. Some algorithms are more likely than others to suffer overfitting, but in general, it is an effect that occurs when the relationship between the number of samples and the number of input variables is low. If the dataset used to train and test the model is not representative of the model's data, the model validation metric is likely to be overestimated, and the performance of the model with the actual data is worse than expected.

Being a problem that is often intrinsic to the data used, the simplest way to prevent it or mitigate its effect is by improving model validation. Techniques such as cross validation are used to deal with this problem.

Some algorithms may use a process called regularization during their training. This process involves modifying the loss function (the function that is attempted to be minimized during training) by adding a parameter that penalizes the generated model's complexity. This parameter's strength is controlled by a parameter that acts as multiplying weight and is often called a regularization parameter or λ .

B.1.3 Model validation through K-Folds

The cross validation is an iterative process. In each iteration or round, the whole dataset is partitioned into two complementary subsets. The first one, called the training data, is used to train the model. The second one called testing data, is used to evaluate the model performance. This process is repeated several times to

reduce the variability, aggregating each round's evaluation metrics (usually with the average) to estimate the right model's performance.

There are several ways to execute a cross validation. Some of them are specifically designed for a specific task. I have used K-Folds Cross Validation in this work. This technique is one of the simplest ways to implement cross validation but also is the most used [129].

K-Folds process starts by creating a partition of the dataset in k , equal size, subsets, then the algorithm is repeated k times. In each iteration, one of the subsets will be designed as the test data, and the other subsets are merged to create the training data. The model is trained with the trained data and evaluated with the testing data. The k evaluation metrics are averaged to calculate the general metric.

Figure B.1 shows an example of a partition of a dataset for $k = 4$. Each of the rows represents one of the iterations performed and displays the testing set and training set.

Figure B.1: K Folds cross validation

In some cases, like working with an unbalanced dataset, a variant of the standard K-Folds is used: the stratified K-Fold. The only difference with the standard version is in the data partition. With stratified K-Folds, the partitions are made, ensuring that the sample's classes' ratio is equal in all partitions. This variation ensures that all partitions will have positive samples in the case of unbalanced binary classification.

When the model to evaluate is not stable, another variation can be used: repeated K-Folds. This version runs the K-Folds several times and averages the individual outputs.

B.2 Machine Learning algorithms

The algorithms described in this section are those used in the development of the thesis. One of the requirements established in Chapter 3 was the scalability of the proposed solutions, for this, it was decided to use the Apache Spark framework, for this reason, the range of algorithms is that offered by Spark’s Machine Learning module [121]. This set of algorithms is extensive and covers most of the most used algorithms today. The exception is neural networks, specifically Deep Learning. These techniques require customization to the problem, prior knowledge, and computing capacity that did not fit the work requirements.

The implementation of these algorithms may vary slightly between different machine learning libraries or modules, but the algorithm’s core will be the same. For each of the algorithms, I list and explain their configuration parameters as they appear in the Apache Spark implementation.

B.2.1 Logistic Regression

Logistic regression (or logit model) is a statistical model which uses a logistic function to model a dependent variable. It is a variation of a Generalized Linear Model [130] that predicts the target variable’s probability. The mathematical expression of the model is represented with the following formula:

$$f(y) = \frac{1}{1 + \exp(\beta_0 + x_1\beta_1 + \dots + x_n\beta_n)} \quad (\text{B.1})$$

Where x is the input features of the model and β the weights assigned to each of them, the training of this model consists of finding the set of values for the weights so that the classification error is minimized. This process is usually treated as an optimization with numerical methods, such as gradient descent [131], so one of these algorithm’s parameters is the number of optimization iterations that will be performed during the training.

One of the advantages of this algorithm versus other Generalized Linear models is that it does not assume a linear relationship between the independent variables (the model’s features) and the target variable. It can work with not regularized variables, and it does not assume homoscedasticity. Its main drawback is that it requires much more data than other algorithms to achieve similar results.

It only has two parameters to be configured:

- *maxIter*: the maximum number of optimization iterations that can be performed in the training process.
- *elasticNetParam*: this is the regularization factor. It takes values between 0 and 1, where 0 means that only the L2 penalty is applied analogously 1 for the L1 penalty. For intermediate values, the regularization is performed with a mix of both penalties.
- *regParam*: this the weight of the regularization penalty to avoid overfitting. It takes a value between 0 and 1.

B.2.2 Support Vector Machine

Support vector machines, or SVM [132], are a type of mathematical algorithm based on the search for a hyperplane capable of separating the data samples according to their category based on the values of the dependent variables. In theory, it is possible to find a hyperplane that works with almost any type of data but finding such a hyperplane can be a task too complex in practice. For this reason, many of the implementations of this algorithm, including the one present in Apache Spark, are limited to looking for a hyperplane linearly dependent on the input variables. The mathematical expression of this hyperplane is the following:

$$0 = b + x_1w_1 + \dots x_nw_n \quad (\text{B.2})$$

Where x is the input variables, w the weight that is assigned to each of these variables, and b a bias parameter, the learning process of this algorithm is to find the set of weights (and variable bias) that separate the different categories of data, maximizing the distance of the samples to the hyperplane.

In general, SVM is an algorithm that can achieve an excellent success rate; however, its linear version may not achieve good results with some datasets. This is because the algorithm assumes that there is a linear relationship between variables, or that they can at least be separated with a linear function. With real systems data, this is not always possible. One of its advantages over other algorithms is that it needs a relatively small number of samples.

There are two parameters to configure in this algorithm:

- *maxIter*: the maximum number of optimization iterations that can be performed in the training process.

- *regParam*: this is the same parameter as the one in Logistic Regression. It takes a value between 0 and 1.

B.2.3 Naive Bayes

Naive Bayes is one of the simplest classification algorithms. Its operation is based on the Bayes theorem, assuming that the input variables are independent. Based on the Bayes theorem, the conditional probability of each of the possible labels is calculated based on the value of the dependent variables, and the largest of them is selected as the output:

$$P(C_i|\vec{x}) = \frac{P(C_i)P(\vec{x}|C_i)}{P(\vec{x})} \quad (\text{B.3})$$

Despite its simplicity, this algorithm has proven to be effective in many jobs [133]–[135]. Its primary disadvantage is that it assumes that variables are independent and that classes have the same dispersion (Homoscedasticity). To solve the problem of working with unbalanced classes, the smoothing parameter is incorporated, α . This parameter is used to weight the probabilities of each of the classes. Originally the probability of each class is calculated as:

$$P(C_i) = \frac{\#C_i}{N} \quad (\text{B.4})$$

Where $\#C_i$ is the number of samples belonging to the class C_i and N the total samples. Adding the smoothing parameter α the equation turns out to be:

$$P(C_i) = \frac{\#C_i + \alpha}{N + d\alpha} \quad (\text{B.5})$$

The smoothing is the only parameter to configure in this algorithm where d is the number of different classes in the data.

- *smoothing*: this is the smoothing weight. It takes a value between 0 and 1.

B.2.4 Decision Tree

The Decision Tree [136] algorithm is based on the tree data structure. The model generated is a tree in which each node is split of the data based on an attribute value. The training process is to determine what will be those split conditions based on the training data. This process is repeated until all samples in the branch have the same value, or the information gained makes another split too low.

The training algorithm looks for the best division, which divides the set into homogeneous subsets with the same value as the target variable. This measure of

division quality is calculated with a measure of the information of the subsets created; the most common forms are entropy and Gini impurity. Gini impurity can be measured as the probability of miscategorizing an element in a set by choosing it randomly.

The Decision Tree is a simple algorithm but with some advantages over other techniques, even more, complex ones. The first one is that the model is easy to understand. This model's core is a decision tree that can be interpreted as a series of if-then conditions. Then, as the decisions are threshold comparisons, there is no assumption of any relation between the target and the features to perform well with nonlinear data. The kind of decisions made in the trees is valid both for continuous and categorical features. Finally, this algorithm has a reduced number of parameters to configure. In some cases, it is considered nonparametric since the few hyperparameters that must be configured do not significantly influence the algorithm's performance, and usually, its values limit memory usage or training time.

One of the problems with decision trees is their high variance. When samples have values close to the thresholds used in tree splits, small changes can lead to large output variations. If the maximum depth of the tree is not limited, the model may suffer from overfitting. Finally, if the model's decision areas are studied, the boundaries are more abrupt than in other algorithms not based on this data structure.

Other algorithms, like Random Forest or Gradient Boosted Trees, rely on these models to create a more complex model made of several Decision Trees aggregated. This algorithm has the following list of parameters:

- *impurity*: the method used to calculate the information gained in the tree splits. It can be *entropy* or *gini*.
- *maxDepth*: maximum depth of the tree. This parameter takes an integer value between 0 and 30. If this parameter is set to a low value, it may not correctly learn the complexity of the training data, affecting the model performance. If it is too high, the model may be overfitted. This also directly impacts the training time because it is directly related to the maximum number of splits searched.

B.2.5 Random Forest

Random Forest [137] is an algorithm that works by aggregating Decision Tree models. The aggregation technique of this algorithm and similar ones is called bagging. With this technique, all models that make up the primary model receive the same input, or in some cases, a limited version of it, and generate a response.

The output of the model is generated by weighting the outputs of the models in the bag. In the case of classification, the output will be the most repeated class. In the case of regression, there are different methods of adding the individual outputs. The simplest of them is that the output of the model is the average of the tree outputs.

Random Forest is a versatile and robust algorithm. This means that it can quickly adapt to different problems and is not too sensitive to data problems such as noise, class imbalance, or overfitting. This is achieved by the mechanism of grouping several models, in this case, decision trees. When training these decision trees, not all of them will use all the data or the same number of variables. This gives the model robustness because if there is a problem with the data, not all the trees would be affected.

It is the algorithm I have used mostly in experiments because it has a good relationship between the performance it offers, its difficulty in training, and its low sensitivity to problems in the data.

The algorithm has several hyperparameters, although it is not very sensitive to the small adjustments made to them. These hyperparameters can change from one implementation to another of the algorithm, although they will generally be similar, the ones I describe to continuation are the ones that can be configured in the Apache Spark MLlib package:

- *numTree*: This is the number of trees that make up the model. The larger it is, the more likely the model will learn more complex patterns, but if it is too high, the case of overfitting can also come. It is also the number of trees to be trained, so it directly relates to training time. It takes an integer value greater or equal to 1 (if it is set to 1, the model will be just a Decision Tree).
- *maxDepth*: the maximum depth that trees can reach. It is the same parameter described in the Decision Tree. It takes an integer value from 0 to 30.
- *featureSubsetStrategy*: in some cases, especially when the number of variables is high, it is interesting that not all input variables be considered in the tree splits. This creates simpler trees, which can detect more general patterns. This variable has a categorical value that indicates the ratio of features each of the trees will receive. In Spark's case, this variable can take the values: *all* - all features, *onethird* - one third of the features, *log2* - logarithm in base 2 of the total number of features, and *sqrt* - square root of the number of features.
- *impurity*: how information gain is measured in each tree decision. It is the same as described in the Decision Tree. It can be *entropy* or *gini*.

B.2.6 Gradient Boosted Tree

Gradient Boosted Trees [138] is an algorithm that, like Random Forest, is made by aggregating Decision Trees. In this case, the aggregation type is boosting [139]. Boosting aggregation is the idea that many weak classifiers can be combined to create a robust classifier. The way to do this is to train each of the classifiers to focus on one part of the data and classify correctly that distinct side of the dataset, the weight of the classifier in the model output will be higher if its individual output is positive. For example, in the case of a dataset like the used in this work, one classifier may be good at predicting an event when a particular pattern appears, and other classifiers would be good to predict the event with other patterns but not with the first one.

This algorithm has more or less the same advantages as Random Forest. The main difference is that it can achieve better results, and it is less prone to overfitting. On the other hand, the computational cost to train this kind of model is several times higher than Random Forest.

The parameters are also quite similar to the ones in Random Forest:

- *maxIter*: this is the number of Decision Trees used in the model. As in the case of Random Forest, the number of trees in the model is proportional to the training time. It takes an integer value with a minimum value of 1 (in that case, the model will be just a Decision Tree).
- *stepSize*: in other algorithms, this parameter is named Learning Rate, usually represented with η . For optimization algorithms, such as gradient descent, it controls the algorithm's advancement in each iteration. It takes a value from 0 to 1. Lower values (< 0.1) generate better classifiers, but it also increases both the training and predicting time.
- *maxDepth*: the maximum depth that trees can reach. It is the same parameter described in the Decision Tree. It takes an integer value from 0 to 30.
- *featureSubsetStrategy*: this controls how many features are taken into account in the tree splits. It is the same parameter as described in Random Forest. Its value is categorical: *all*, *onethird*, *log2*, or *sqrt*.
- *impurity*: how information gain is measured in each tree decision. It is the same as described in the Decision Tree. The current implementation of this algorithm in Apache Spark only allows *gini*.

B.3 Data Processing techniques

B.3.1 KDE

Kernel Density Estimation (KDE) [140] is an algorithm from the family of density estimators. These algorithms try to model the probability function that generates a dataset. This function can be used to create simple but powerful simulations. KDE is a non parametric statistic process. It can be understood as the extension of a discrete histogram to a continuous domain.

The process to calculate the estimation is based on replacing each point in the distribution to estimate by the kernel function. Let $(x_1, x_2, \dots, x_n)$ be an independent and identically distributed sample with n observations obtained from an unknown probability function $f(x)$. The kernel estimation $\hat{f}(x)$ is calculated as:

$$\hat{f}(x) = \frac{1}{n} \sum_{i=1}^n K(x_i, t) \quad (\text{B.6})$$

Where $K(x, t)$ is the Kernel function selected, this kernel function may vary from one application to another. The most common ones are uniform (rectangular or tophat), triangular, biweight, triweight, Epanechnikov, normal (or gaussian).

B.4 Agent Computing

Agent Computing is a programming paradigm based on the concept of Software Agent [141]. A software agent, also called softbot, bot, or actor, is a computer program capable of acting rigorously to accomplish a specific task. The main difference between an agent with a regular computer program is that the agents are designed to be autonomous, without the need for human interaction, and usually with capabilities to communicate with other agents. Software agents are a programming abstraction implemented in many frameworks, which allows designing the agent focusing on the task that it must solve, ignoring the communication protocol or the application's life cycle.

Agents or actors are independent processes capable of making decisions, usually in a small part of the problem to be solved. Actors are autonomous in decision-making but can exchange information with other actors through a message-based communication mechanism. By combining multiple simple actors, you can achieve a complex system capable of solving big problems. The agent's concept is old, initially proposed in [142], and has adapted to different technologies and challenges.

Appendix C

Detailed results

C.1 Scenario

ev	F	TF	P	S		ev	F	TF	P	S		ev	F	TF	P	S	
0	1208	1194	1097	97	8%	131	14	14	14	0	0%	263	5	4	4	0	0%
1	656	650	634	16	2%	132	14	14	14	0	0%	264	5	5	5	0	0%
2	633	632	523	109	17%	133	14	14	9	5	35%	265	5	5	5	0	0%
3	608	460	455	5	1%	134	13	13	12	1	7%	266	5	4	4	0	0%
4	605	605	483	122	20%	135	13	13	13	0	0%	267	5	5	5	0	0%
5	602	466	464	2	0%	136	13	13	13	0	0%	268	5	5	5	0	0%
6	466	46	46	0	0%	137	13	13	12	1	7%	269	5	4	4	0	0%
7	466	46	46	0	0%	138	13	13	10	3	23%	270	5	5	5	0	0%
8	440	144	140	4	2%	139	13	13	13	0	0%	271	5	5	5	0	0%
9	438	144	142	2	1%	140	13	13	12	1	7%	272	5	5	5	0	0%
10	430	131	123	8	6%	141	13	13	12	1	7%	273	5	5	5	0	0%
11	428	39	39	0	0%	142	13	13	13	0	0%	274	5	5	5	0	0%
12	428	39	39	0	0%	143	13	13	13	0	0%	275	5	5	5	0	0%
13	425	124	123	1	0%	144	12	12	11	1	8%	276	5	5	5	0	0%
14	367	367	299	68	18%	145	12	12	12	0	0%	277	4	4	4	0	0%
15	352	352	350	2	0%	146	12	12	12	0	0%	278	4	4	4	0	0%
16	313	312	310	2	0%	147	12	12	12	0	0%	279	4	1	1	0	0%
17	298	298	297	1	0%	148	12	12	9	3	25%	280	4	4	4	0	0%
18	289	289	280	9	3%	149	12	12	12	0	0%	281	4	4	4	0	0%
19	289	88	85	3	3%	150	12	12	12	0	0%	282	4	4	4	0	0%
20	288	86	86	0	0%	151	12	12	12	0	0%	283	4	4	4	0	0%
21	279	76	76	0	0%	152	12	12	12	0	0%	284	4	4	4	0	0%
22	279	279	177	102	36%	153	12	12	12	0	0%	285	4	4	3	1	25%
23	278	82	82	0	0%	154	11	11	11	0	0%	286	4	2	1	1	50%
24	272	249	249	0	0%	155	11	11	11	0	0%	287	4	4	4	0	0%
25	272	248	230	18	7%	156	11	11	10	1	9%	288	4	4	3	1	25%
26	272	248	230	18	7%	157	11	11	11	0	0%	289	4	4	4	0	0%
27	272	249	249	0	0%	158	11	11	11	0	0%	290	4	4	4	0	0%
28	266	266	263	3	1%	159	11	11	11	0	0%	291	4	4	4	0	0%
29	265	265	263	2	0%	160	11	11	10	1	9%	292	4	4	4	0	0%
30	241	241	240	1	0%	161	11	11	11	0	0%	293	4	4	4	0	0%
31	239	239	239	0	0%	162	10	10	10	0	0%	294	4	4	4	0	0%
32	221	220	220	0	0%	163	10	10	10	0	0%	295	4	4	4	0	0%
33	214	214	213	1	0%	164	10	10	10	0	0%	296	4	4	4	0	0%
34	214	214	214	0	0%	165	10	6	6	0	0%	297	4	4	4	0	0%
35	211	210	195	15	7%	166	10	10	10	0	0%	298	4	2	1	1	50%
36	211	209	209	0	0%	167	10	10	10	0	0%	299	4	4	4	0	0%
37	177	177	161	16	9%	168	10	10	10	0	0%	300	4	4	4	0	0%
38	155	66	65	1	1%	169	10	10	8	2	20%	301	4	4	4	0	0%
39	151	55	55	0	0%	170	10	10	10	0	0%	302	3	3	2	1	33%
40	149	37	37	0	0%	171	10	10	10	0	0%	303	3	1	1	0	0%
41	139	66	65	1	1%	172	10	10	10	0	0%	304	3	3	2	1	33%
42	137	58	58	0	0%	173	10	10	8	2	20%	305	3	3	3	0	0%
43	136	66	65	1	1%	174	10	10	10	0	0%	306	3	3	3	0	0%
44	131	34	34	0	0%	175	10	10	10	0	0%	307	3	3	3	0	0%
45	130	32	32	0	0%	176	10	10	10	0	0%	308	3	3	3	0	0%
46	130	56	55	1	1%	177	10	10	10	0	0%	309	3	3	3	0	0%
47	124	28	28	0	0%	178	10	10	10	0	0%	310	3	3	3	0	0%
48	123	123	123	0	0%	179	10	10	10	0	0%	311	3	3	3	0	0%
49	122	122	113	9	7%	180	9	9	6	3	33%	312	3	3	3	0	0%
50	122	120	120	0	0%	181	9	9	9	0	0%	313	3	3	3	0	0%

51	121	120	120	0	0%	182	9	9	9	0	0%	314	3	3	2	1	33%
52	120	120	120	0	0%	183	9	9	9	0	0%	315	3	3	3	0	0%
53	117	117	116	1	0%	184	9	7	6	1	14%	316	3	3	3	0	0%
54	116	116	115	1	0%	185	9	9	9	0	0%	317	3	3	3	0	0%
55	116	83	83	0	0%	186	9	9	9	0	0%	318	3	3	3	0	0%
56	110	110	109	1	0%	187	9	9	9	0	0%	319	3	3	2	1	33%
57	109	93	92	1	1%	188	9	9	9	0	0%	320	3	3	3	0	0%
58	106	64	64	0	0%	189	9	9	9	0	0%	321	2	2	2	0	0%
59	95	10	10	0	0%	190	9	6	5	1	16%	322	2	2	2	0	0%
60	95	10	10	0	0%	191	9	9	9	0	0%	323	2	2	2	0	0%
61	95	95	95	0	0%	192	9	9	9	0	0%	324	2	2	2	0	0%
62	94	9	9	0	0%	193	9	5	5	0	0%	325	2	2	2	0	0%
63	94	9	9	0	0%	194	8	8	8	0	0%	326	2	2	2	0	0%
64	94	10	10	0	0%	195	8	8	8	0	0%	327	2	2	2	0	0%
65	94	10	10	0	0%	196	8	8	8	0	0%	328	2	2	2	0	0%
66	93	9	9	0	0%	197	8	8	5	3	37%	329	2	2	2	0	0%
67	93	9	9	0	0%	198	8	8	8	0	0%	330	2	2	2	0	0%
68	83	26	26	0	0%	199	8	8	8	0	0%	331	2	2	2	0	0%
69	75	57	57	0	0%	200	8	8	5	3	37%	332	2	2	2	0	0%
70	74	33	33	0	0%	201	8	8	8	0	0%	333	2	2	2	0	0%
71	71	32	32	0	0%	202	8	8	8	0	0%	334	2	2	2	0	0%
72	70	34	34	0	0%	203	8	8	8	0	0%	335	2	2	1	1	50%
73	63	40	39	1	2%	204	8	8	8	0	0%	336	2	2	2	0	0%
74	63	45	45	0	0%	205	8	8	8	0	0%	337	2	2	2	0	0%
75	56	56	56	0	0%	206	8	8	8	0	0%	338	2	2	2	0	0%
76	54	54	53	1	1%	207	8	8	8	0	0%	339	2	2	1	1	50%
77	51	15	14	1	6%	208	8	8	8	0	0%	340	2	2	2	0	0%
78	50	35	35	0	0%	209	8	8	8	0	0%	341	2	2	2	0	0%
79	50	50	49	1	2%	210	8	8	8	0	0%	342	2	2	2	0	0%
80	50	50	49	1	2%	211	7	7	7	0	0%	343	2	2	2	0	0%
81	49	33	33	0	0%	212	7	7	7	0	0%	344	2	2	2	0	0%
82	48	48	43	5	10%	213	7	7	7	0	0%	345	2	2	2	0	0%
83	47	14	14	0	0%	214	7	7	6	1	14%	346	2	2	1	1	50%
84	46	40	39	1	2%	215	7	7	6	1	14%	347	2	2	2	0	0%
85	46	46	45	1	2%	216	7	7	7	0	0%	348	2	2	2	0	0%
86	46	46	45	1	2%	217	7	7	7	0	0%	349	2	2	2	0	0%
87	45	12	10	2	16%	218	7	7	6	1	14%	350	2	2	2	0	0%
88	44	30	30	0	0%	219	7	7	7	0	0%	351	2	2	2	0	0%
89	43	12	12	0	0%	220	7	7	7	0	0%	352	2	2	2	0	0%
90	39	36	36	0	0%	221	7	7	6	1	14%	353	2	2	2	0	0%
91	29	13	13	0	0%	222	7	7	7	0	0%	354	2	2	2	0	0%
92	27	20	20	0	0%	223	7	7	7	0	0%	355	2	2	1	1	50%
93	25	25	24	1	4%	224	7	7	7	0	0%	356	2	2	2	0	0%
94	24	18	18	0	0%	225	6	6	6	0	0%	357	2	2	2	0	0%
95	24	16	16	0	0%	226	6	6	6	0	0%	358	2	2	2	0	0%
96	23	23	23	0	0%	227	6	6	5	1	16%	359	2	2	2	0	0%
97	23	23	22	1	4%	228	6	6	5	1	16%	360	2	2	2	0	0%
98	22	7	7	0	0%	229	6	6	6	0	0%	361	2	2	2	0	0%
99	22	22	22	0	0%	230	6	6	6	0	0%	362	2	2	1	1	50%
100	22	6	6	0	0%	231	6	6	6	0	0%	363	2	2	2	0	0%
101	22	11	11	0	0%	232	6	6	6	0	0%	364	2	2	2	0	0%
102	22	6	6	0	0%	233	6	6	6	0	0%	365	2	2	2	0	0%
103	22	22	22	0	0%	234	6	6	5	1	16%	366	2	2	2	0	0%
104	21	21	20	1	4%	235	6	6	6	0	0%	367	2	2	2	0	0%
105	21	21	21	0	0%	236	6	6	6	0	0%	368	2	2	2	0	0%
106	21	21	20	1	4%	237	6	6	6	0	0%	369	2	2	2	0	0%
107	21	21	21	0	0%	238	6	6	5	1	16%	370	2	2	2	0	0%
108	21	21	21	0	0%	239	6	6	5	1	16%	371	2	2	2	0	0%
109	20	10	10	0	0%	240	6	6	6	0	0%	372	2	2	2	0	0%
110	20	20	20	0	0%	241	6	6	6	0	0%	373	2	2	2	0	0%
111	20	20	19	1	5%	242	6	6	5	1	16%	374	2	2	1	1	50%
112	20	20	20	0	0%	243	5	5	5	0	0%	375	2	2	2	0	0%
113	20	16	16	0	0%	244	5	5	5	0	0%	376	2	2	2	0	0%
114	20	20	20	0	0%	245	5	5	3	2	40%	377	2	2	2	0	0%
115	19	12	12	0	0%	246	5	4	4	0	0%	378	2	1	1	0	0%
116	19	12	12	0	0%	247	5	5	5	0	0%	379	2	2	2	0	0%
117	19	2	2	0	0%	249	5	5	5	0	0%	380	2	2	2	0	0%
118	19	2	2	0	0%	250	5	5	5	0	0%	381	2	2	2	0	0%
119	18	18	18	0	0%	251	5	5	5	0	0%	382	2	2	1	1	50%
120	18	18	18	0	0%	252	5	4	4	0	0%	383	2	2	2	0	0%
121	18	18	18	0	0%	253	5	5	5	0	0%	384	2	2	2	0	0%
122	17	10	10	0	0%	254	5	5	4	1	20%	385	2	2	2	0	0%
123	17	16	16	0	0%	255	5	5	5	0	0%	386	2	2	2	0	0%
124	17	17	17	0	0%	256	5	5	5	0	0%	387	2	1	1	0	0%
125	16	15	10	5	33%	257	5	4	4	0	0%	388	2	2	1	1	50%
126	16	8	8	0	0%	258	5	5	5	0	0%	389	2	2	2	0	0%
127	15	15	15	0	0%	259	5	5	5	0	0%	390	2	2	2	0	0%
128	15	14	13	1	7%	260	5	5	5	0	0%	391	1	1	1	0	0%
129	15	5	5	0	0%	261	5	5	5	0	0%	392	1	1	1	0	0%
130	14	14	11	3	21%	262	5	5	5	0	0%	393	1	1	1	0	0%

Table C.1: Event frequencies

ev: event, **F**: Frequency, **TF**: True Frequency, **P**: Predictable, **S**: Spontaneous

C.2 Methodology

C.2.1 Step 1

Event	F-Score	Event	F-Score	Event	F-Score	Event	F-Score
0	0.02441008	29	0.12658654	54	0.07698179	90	0.08444444
1	0.00610687	30	0.07438468	55	0.37498317	95	0
2	0.21411188	31	0.06774497	57	0.06533333	99	0.11428571
3	0.26676933	32	0.04088889	58	0.21178915	103	0.65968254
5	0.27880009	33	0.1667914	59	0.43333333	106	0
6	0.06493506	34	0.5074159	60	0.43333333	107	0.41666667
7	0.06493506	35	0.1212461	61	0.33475751	108	0
8	0.33354812	36	0.34306958	68	0.10714286	112	0.63714286
9	0.30282298	37	0	69	0	113	0.08
10	0.11231982	38	0.68655551	70	0.07272727	116	0.36666667
11	0	39	0.48349206	71	0.18444444	120	0.14666667
12	0	40	0.11076923	72	0.115	123	0.19111111
13	0.21197765	41	0.51480331	73	0.11636364	124	0.2
14	0.13000883	42	0.60877617	74	0.13131313	127	0.33333333
15	0.19195682	43	0.456298	75	0.31604278	128	0
16	0.24771489	44	0.04	76	0.16240602	131	0.34666667
17	0.63074152	45	0.21675214	78	0.13444444	132	0.2
21	0.01666667	46	0.60731551	79	0	133	0
23	0	47	0.05	80	0	136	0.16666667
24	0.61193266	48	0.11966845	81	0.03636364	139	0
25	0.79632695	50	0.1416731	84	0	142	0.16666667
26	0.79484208	51	0.14502332	85	0	145	0
27	0.61562441	52	0.06548265	86	0	153	0.26666667
28	0.13135863	53	0.08407888	88	0.31575758		

Table C.2: F-Score results of step 1

C.2.2 Step 2

EV	ET	Gen	FS	LT	OW	EV	ET	Gen	FS	LT	OW
0	48	32	0.65	60	237.95	90	149	75	0.33	248	11.67
1	156	75	0.05	1437	8.67	91	2	1	0	60	22.51
2	38	20	0.35	60	50.64	92	2	1	0	92	0.17
3	133	63	0.35	67	7.20	93	2	1	0	602	0.17
4	10	5	0	60	0.17	94	1	1	0	570	0.17
5	50	26	0.36	576	55.05	95	37	20	0.36	471	0.17
6	47	23	0.32	163	0.17	96	2	1	0	312	2.98
7	29	15	0.23	256	2.38	97	2	1	0	60	18.26
8	36	20	0.74	345	50.18	98	35	22	0.6	184	0.17
9	96	50	0.63	158	18.12	99	104	1	0.41	60	76.18
10	129	63	0.29	294	0.17	100	26	18	0.5	924	8.75
11	78	38	0.32	157	0.17	101	1	1	0	799	0.17
12	71	35	0.34	153	0.17	102	4	3	0	60	0.17
13	82	43	0.66	598	0.17	103	38	19	0.77	291	0.17
14	32	17	0.24	90	63.86	104	2	1	0	572	12.84
15	48	23	0.73	582	1.00	105	2	1	0	106	0.17
16	81	40	0.38	328	16.73	106	36	18	0.34	177	0.17
17	29	15	0.8	60	16.77	107	50	40	0.58	481	0.17
18	2	1	0	60	6.48	108	130	69	0.26	1122	0.17
19	2	1	0	60	0.17	109	1	1	0	96	20.69
20	11	6	0	284	0.17	110	2	1	0	322	0.17
21	85	44	0.57	599	0.17	111	2	1	0	904	0.17
22	2	1	0	641	0.17	112	45	23	0.7	174	2.13
23	54	28	0.47	595	0.17	113	141	75	0.28	336	0.17
24	36	22	0.79	808	56.18	114	2	1	0	60	0.17
25	45	23	0.81	78	3.15	115	1	1	0	181	21.75
26	50	25	0.8	75	3.37	116	81	70	0.89	60	172.53
27	21	13	0.78	604	65.18	117	1	1	0	200	14.32
28	33	20	0.25	60	78.42	118	1	1	0	60	8.05
29	75	39	0.21	60	12.58	119	2	1	0	318	14.79
30	147	75	0.13	131	20.67	120	158	1	0.45	114	1.47
31	24	13	0.15	193	39.42	121	2	1	0	518	20.22
32	50	29	0.09	60	39.02	122	1	1	0	60	0.17

33	73	43	0.33	239	54.33	123	106	66	0.7	275	1.33
34	54	30	0.7	419	16.75	124	37	19	0.2	72	0.17
35	139	75	0.24	261	24.15	125	2	1	0	80	0.17
36	138	75	0.39	372	23.47	126	1	1	0	60	0.17
37	24	14	0.09	60	55.46	127	98	71	0.56	546	3.15
38	32	18	0.75	60	9.50	128	108	75	0.41	621	51.78
39	99	75	0.91	411	71.52	129	36	23	0.6	218	0.17
40	39	26	0.48	216	87.65	130	2	1	0	285	11.55
41	48	26	0.65	60	8.13	131	37	19	0.75	294	0.17
42	72	39	0.69	60	6.68	132	24	13	0.77	297	0.17
43	144	75	0.63	63	2.78	133	92	56	0.31	226	1.02
44	64	32	0.26	274	0.17	134	1	1	0	426	0.17
45	149	75	0.43	60	9.57	135	1	1	0	60	0.17
46	59	33	0.7	60	4.02	136	51	27	0.63	593	0.17
47	145	75	0.28	158	8.17	137	1	1	0	674	0.17
48	27	15	0.26	521	0.17	138	2	1	0	60	20.77
49	55	27	0.05	434	11.75	139	29	17	0.2	519	0.17
50	151	75	0.28	523	0.17	140	1	1	0	404	35.58
51	42	22	0.26	596	5.78	141	2	1	0	60	6.39
52	161	75	0.18	60	2.10	142	44	24	0.63	594	0.17
53	145	75	0.23	147	33.05	143	1	1	0	813	0.17
54	80	42	0.17	60	1.28	144	2	1	0	773	0.17
55	85	49	0.49	60	65.55	145	56	29	0.23	1171	2.37
56	2	1	0	417	8.57	146	1	1	0	60	8.50
57	71	39	0.34	1395	44.40	147	1	1	0	573	0.17
58	58	29	0.61	253	5.48	148	2	1	0	107	9.23
59	24	13	0.57	179	5.07	149	1	1	0	284	0.17
60	34	19	0.5	181	9.47	150	1	1	0	60	0.17
61	66	34	0.44	60	6.62	151	1	1	0	421	10.85
62	3	2	0	362	0.17	152	1	1	0	60	23.31
63	1	1	0	60	16.32	153	50	28	0.85	299	0.17
64	1	1	0	60	0.17	154	1	1	0	361	0.17
65	1	1	0	60	17.35	155	1	1	0	356	19.46
66	1	1	0	60	12.46	156	2	1	0	157	0.17
67	1	1	0	60	0.17	157	1	1	0	60	0.17
68	81	64	0.71	1211	174.75	158	1	1	0	636	2.90
69	101	1	0.42	61	64.30	159	1	1	0	161	19.80
70	32	18	0.29	725	0.17	160	2	1	0	166	0.17
71	63	35	0.53	412	22.18	161	1	1	0	60	0.17
72	82	42	0.32	894	0.17	162	1	1	0	488	21.06
73	29	16	0.48	558	29.05	163	2	1	0	60	0.17
74	102	56	0.33	765	33.12	164	1	1	0	469	5.27
75	59	33	0.5	369	11.91	165	46	27	0.73	521	4.72
76	101	54	0.36	85	3.17	166	1	1	0	441	0.17
77	2	1	0	180	0.17	167	1	1	0	785	0.17
78	153	75	0.36	60	7.10	168	1	1	0	60	3.27
79	157	75	0.27	92	0.17	169	2	1	0	453	0.17
80	65	31	0.21	104	0.17	170	1	1	0	60	0.17
81	117	75	0.38	119	10.45	171	1	1	0	961	0.17
82	73	46	0.07	60	80.17	172	1	1	0	60	0.17
83	2	1	0	372	0.17	173	2	1	0	225	0.17
84	115	58	0.19	732	0.17	174	1	1	0	732	0.17
85	157	75	0.2	103	0.17	175	2	1	0	60	6.54
86	25	13	0.08	626	0.17	176	1	1	0	60	7.80
87	2	1	0	164	0.17	177	1	1	0	200	22.34
88	93	75	0.68	365	56.48	178	1	1	0	821	0.17
89	1	1	0	109	5.83	179	2	1	0	417	9.15

Table C.3: Genetic Algorithm optimization results in step 2

ev: event, **ET:** Execution time (min), **Gen:** GA generations

FS: Final F-Score, **LT:** Lead time (s), **OW:** Observation window (min)

C.2.3 Step 3

event	dt	gbt	nb	rf	svm	event	dt	gbt	nb	rf	svm
0	0.6173	0.6447	0.2778	0.6352	0.1923	54	0.1667	0.2634	0.3065	0.1627	0.1670
1	0.0387	0.0535	0.0071	0.0475	0.0453	55	0.4502	0.4252	0.1192	0.4384	0.1053
2	0.3359	0.3701	0.2348	0.3551	0.2340	57	0.2678	0.3481	0.0994	0.3201	0.1376
3	0.3783	0.4134	0.3875	0.3329	0.2782	58	0.5477	0.6405	0.3985	0.5788	0.4676
5	0.2563	0.5019	0.2058	0.3241	0.2504	59	0.5487	0.5422	0.2150	0.5422	0.4353
6	0.2787	0.2781	0.2518	0.3102	0.2806	60	0.4527	0.4756	0.1430	0.4222	0.3293
7	0.2238	0.2241	0.1184	0.2243	0.0603	61	0.4271	0.5257	0.3560	0.4855	0.3900
8	0.7241	0.6932	0.2244	0.7148	0.6320	68	0.5692	0.5925	0.0988	0.6270	0.2586
9	0.6324	0.6329	0.4454	0.6416	0.5635	69	0.3646	0.3535	0.1399	0.3875	0.0570
10	0.2822	0.2420	0.1664	0.2683	0.2419	70	0.2305	0.2873	0.1485	0.2862	0.2383
11	0.2741	0.3048	0.2968	0.3000	0.3183	71	0.5539	0.5555	0.2379	0.5488	0.2489
12	0.2415	0.2933	0.2145	0.3034	0.2746	72	0.2573	0.2839	0.1894	0.2750	0.2505
13	0.6579	0.6648	0.4413	0.6550	0.6235	73	0.4289	0.4921	0.3006	0.4464	0.2843
14	0.2106	0.2789	0.1874	0.2219	0.1091	74	0.2913	0.3405	0.3146	0.3201	0.2748
15	0.7350	0.7348	0.7141	0.7348	0.7330	75	0.4282	0.5004	0.3045	0.4933	0.3844
16	0.3724	0.3831	0.2744	0.3748	0.2651	76	0.3356	0.3906	0.2914	0.3459	0.2414
17	0.8057	0.7944	0.6406	0.8020	0.7767	78	0.2407	0.3486	0.0984	0.2859	0.0567
21	0.6140	0.6239	0.4515	0.5999	0.6046	79	0.1797	0.2804	0.2404	0.1701	0.1200
23	0.4665	0.4828	0.4270	0.4522	0.5435	80	0.1822	0.2159	0.1764	0.1869	0.1738
24	0.7949	0.7791	0.5497	0.7755	0.7480	81	0.2808	0.3922	0.2426	0.3409	0.2249
25	0.7944	0.7958	0.7426	0.7961	0.7008	84	0.2042	0.2270	0.2461	0.1998	0.1834
26	0.7925	0.7936	0.7371	0.7895	0.6635	85	0.1881	0.2488	0.1852	0.1945	0.1056
27	0.7939	0.7800	0.5203	0.7728	0.7531	86	0.0706	0.0788	0.0535	0.0788	0.0713
28	0.2463	0.2526	0.0931	0.2437	0.0644	88	0.5778	0.6279	0.3111	0.6065	0.3507
29	0.1776	0.2031	0.1408	0.1946	0.0820	90	0.2450	0.3201	0.2763	0.2703	0.1402
30	0.1126	0.1415	0.0926	0.1220	0.0514	95	0.2888	0.2492	0.1724	0.2911	0.1714
31	0.1239	0.1311	0.0926	0.1295	0.0539	99	0.2686	0.3584	0.1179	0.3599	0.0445
32	0.0868	0.1205	0.0961	0.0927	0.0369	103	0.7565	0.7701	0.6141	0.7701	0.7565
33	0.3203	0.3272	0.2764	0.3321	0.2254	106	0.2285	0.3118	0.1533	0.2752	0.3402
34	0.6944	0.6835	0.4461	0.6925	0.6413	107	0.5535	0.5473	0.3192	0.5473	0.5243
35	0.1534	0.3960	0.2814	0.2126	0.2035	108	0.2967	0.2792	0.1159	0.2792	0.2433
36	0.3461	0.4939	0.3267	0.3426	0.3322	112	0.6892	0.7097	0.3506	0.7097	0.6671
37	0.0758	0.1226	0.1122	0.0880	0.0406	113	0.1890	0.2781	0.1304	0.2422	0.2118
38	0.6771	0.7153	0.3818	0.7118	0.5653	116	0.6735	0.8316	0.0523	0.7573	0.3760
39	0.8656	0.9071	0.5331	0.8968	0.6174	120	0.5662	0.5505	0.2599	0.5505	0.5797
40	0.4799	0.4602	0.2151	0.5077	0.2911	123	0.6768	0.6827	0.4077	0.6827	0.6622
41	0.5928	0.6491	0.3891	0.6356	0.5089	124	0.1240	0.1267	0.0753	0.1267	0.1262
42	0.6331	0.6694	0.3717	0.6843	0.5576	127	0.4104	0.3567	0.1916	0.4083	0.3648
43	0.5832	0.6171	0.4017	0.6196	0.5224	128	0.3741	0.3800	0.0592	0.3800	0.3411
44	0.1716	0.2838	0.2256	0.1837	0.2273	131	0.7432	0.7505	0.4377	0.7505	0.7332
45	0.3537	0.4020	0.2160	0.4290	0.3056	132	0.7374	0.7187	0.3410	0.7187	0.7174
46	0.6527	0.6763	0.3829	0.6709	0.6080	133	0.3368	0.3933	0.2161	0.3933	0.3688
47	0.2598	0.3270	0.1851	0.2803	0.1599	136	0.5833	0.5711	0.3305	0.5711	0.5833
48	0.2518	0.2884	0.2238	0.2604	0.2486	139	0.2033	0.1933	0.1359	0.2000	0.2020
50	0.2784	0.2854	0.1590	0.2792	0.2828	142	0.5833	0.5711	0.2941	0.5711	0.5833
51	0.1986	0.2952	0.2770	0.2277	0.1668	145	0.1867	0.1556	0.0851	0.1556	0.1867
52	0.1531	0.2199	0.2840	0.1402	0.1776	153	0.8590	0.8594	0.5449	0.8594	0.8390
53	0.1712	0.2317	0.1529	0.1988	0.0752						

Table C.5: Step 3: model selection F-score

dt: Decision Tree, **gbt**: Gradient Boosted Tree, **nb**: Naive Bayes
rf: Random Forest, **svm**: Support Vector Machine

event	dt	gbt	nb	rf	svm	event	dt	gbt	nb	rf	svm
0	1.6	22.7	1	5.3	7	54	2.5	34.5	1	10.9	3.4
1	2.3	30.4	1	9.5	3.1	55	2.7	43.7	1	10	2.8
2	2.5	37.6	1	10.5	3.6	57	2.4	33.7	1	8.4	2.7
3	2.9	41.7	1	13	3	58	2.5	35.6	1	10.6	3.8
5	2.5	34.9	1	9.9	4.2	59	2.5	35.8	1	9.9	1.9
6	1.6	19.1	3.2	5.7	4.8	60	2.5	37.2	1	9.6	1.4
7	2.3	30.7	1	10.3	3.8	61	2.8	38.6	1	11.7	2
8	2.7	35	1	9.5	3.2	68	1.3	13.5	1	3.8	1.8
9	2.4	33.8	1	9	3.2	69	0.9	13.8	1	2.1	3.2
10	1.9	22.8	1	7.4	3	70	1.1	12.6	1	3.7	2.8
11	1.8	20.4	1	6.6	3.6	71	2.1	31.2	1	8.2	3.1
12	1.8	20.9	3.4	7.1	4.5	72	1.5	16.7	1	5.2	3
13	1.5	18.1	1	5.6	3.6	73	2.5	35.4	1	9.5	2.7
14	2.4	31.6	1	9.4	4.3	74	2.4	35.1	3.2	9.1	5.1
15	2.2	27.8	1	9.1	3	75	2.2	31.3	3.2	8.3	5.3
16	2.5	35.4	1	10.1	2.8	76	2.6	34.5	3.2	11.6	4.4
17	2.6	36.4	1	10.3	2.1	78	2.7	40.1	1	11.3	2.1
21	1.6	18.8	1	5.7	3.2	79	1.7	20.5	1	6.6	2.4
23	1.5	17.9	1	5.4	2.3	80	1.8	21.8	1	6.9	2.7
24	1.9	28	1	6.4	2.1	81	2.4	38.9	1	8.9	3.2
25	2.5	33.8	1	10.9	2.9	84	1.3	13	3.4	4.1	5.4
26	2.5	34.6	1	11.5	2.9	85	1.8	21.8	1	7	3.6
27	1.8	25.1	1	5.6	2	86	1.3	14.9	1	4.2	1.9
28	2.3	36.9	3.3	7.9	5	88	1	12.1	1	2.9	0.8
29	2.7	41.7	1	11.7	2.7	90	2.3	34.7	1	8.9	3
30	2.5	37.2	1	10	2.2	95	1.4	16.3	1	4.8	2.6
31	2.5	38	1	10.2	3	99	1.7	27.7	1	4.4	2.1
32	2.5	38.1	1	10.1	3	103	1.7	20.2	1	6.4	2.6
33	2.4	35.2	1	8.2	3.9	106	1.6	19.8	1	6.1	2.7
34	2.1	29.2	1	6.3	3.1	107	1.6	18.6	1	5.5	2
35	2.3	34.9	1	8.8	2.6	108	1.2	14.9	1	3.9	2.8
36	2.2	31.3	1	8.2	2.4	112	2.4	32.4	1	10.2	2.2
37	2.3	33.6	1	8.3	2.7	113	1.4	15.9	1	4.9	2.5
38	2.7	40.7	3.2	11.1	5.6	116	1.2	12.1	1	3.2	0.7
39	1.2	15.4	1	3.4	3.4	120	2.4	31.1	1	10.7	2.5
40	1.6	34.4	1	5	2.4	123	2.1	28.7	3.2	9.5	3.9
41	2.8	40.6	3.2	12.2	4.8	124	1.9	22.8	1	7.4	3
42	2.4	40.2	1	9.8	4.1	127	0.8	14	1	2	0.7
43	2.4	38.2	1	10.4	3.2	128	2	26	3.2	5.1	4.1
44	1.5	18.1	1	5.5	3.2	131	1.9	22.5	1	6.9	1.9
45	2.7	40.4	1	11	2.6	132	1.9	22.6	1	6.9	1.7
46	2.4	36.4	1	8.3	2.4	133	2.1	26.7	1	8.7	2
47	2.7	39.1	1	11.3	3	136	1.5	17.9	3.2	5.4	3.5
48	1.4	16.7	1	4.5	2.8	139	1.3	16.8	1	4.3	2.5
50	1.5	17.5	1	4.8	3.4	142	1.5	18	1	5.3	1.5
51	2.3	31.1	1	9.2	2.5	145	2.1	27.1	1	8.2	1.6
52	2.7	36.7	1	11.8	2.7	153	1.7	21.1	1	5.6	1.9
53	2.7	36.6	1	9.8	3.2						

Table C.7: Step 3: model selection times

dt: Decision Tree, **gbt**: Gradient Boosted Tree, **nb**: Naive Bayes
rf: Random Forest, **svm**: Support Vector Machine

C.2.4 Step 4

event	Features	F-Score	event	Features	F-Score
0	196	0.6327	54	548	0.3065
1	302	0.0354	55	190	0.4239
2	86	0.3586	57	44	0.2833
3	5	0.2864	58	261	0.6008
5	102	0.4888	59	280	0.6000
6	5	0.3519	60	5	0.5000
7	46	0.1907	61	28	0.5192
8	12	0.7422	68	21	0.6054
9	294	0.6575	69	22	0.3415
10	357	0.3380	70	29	0.2572
11	145	0.3315	71	173	0.5529
12	5	0.3422	72	181	0.3008
13	10	0.7223	73	365	0.3580
14	46	0.3018	74	217	0.2982
15	5	0.7494	75	7	0.5452
16	400	0.3739	76	103	0.2206
17	200	0.8896	78	196	0.2486

21	25	0.6160	79	197	0.1969
23	140	0.5114	80	49	0.1963
24	5	0.9757	81	50	0.3080
25	139	0.8743	84	9	0.2461
26	97	0.8718	85	193	0.1972
27	5	0.9069	86	61	0.0532
28	438	0.2302	88	114	0.5120
29	498	0.1843	90	166	0.2363
30	19	0.1392	95	90	0.2515
31	175	0.1333	99	71	0.2533
32	66	0.0653	103	5	0.7520
33	89	0.3162	106	10	0.3951
34	18	0.7408	107	407	0.4326
35	5	0.2053	108	272	0.0989
36	38	0.4070	112	540	0.7299
37	38	0.0761	113	5	0.2195
38	29	0.7333	116	166	0.7841
39	136	0.6902	120	6	0.5024
40	276	0.4564	123	305	0.7046
41	68	0.7013	124	5	0.1633
42	46	0.6752	127	241	0.3531
43	195	0.5579	128	36	0.7470
44	34	0.2857	131	5	0.7923
45	105	0.3907	132	5	0.7814
46	35	0.6649	133	187	0.3313
47	446	0.2580	136	5	0.5693
48	8	0.2845	139	380	0.1603
50	21	0.5071	142	340	0.5700
51	17	0.2730	145	5	0.2203
52	548	0.2840	153	178	0.8254
53	167	0.0891			

Table C.8: Step 4: number of selected features

C.2.5 Step 5

event	sampler	Positive samples	F-Score	event	sampler	Positive samples	F-Score
0	adasyn	9053	0.6327	54	allknn	116	0.3065
1	cc	650	0.1289	55	adasyn	10151	0.4239
2	smote enn	3414	0.3965	57	allknn	93	0.3688
3	rus	460	0.4834	58	adasyn	10334	0.6661
5	allknn	466	0.4888	59	smote	7812	0.6000
6	ros	298	0.3519	60	ros	22	0.5000
7	allknn	46	0.2568	61	smote	5341	0.5805
8	adasyn	2157	0.8093	68	allknn	26	0.6054
9	smote enn	5840	0.6575	69	ros	3647	0.3458
10	smote tomek	10397	0.3594	70	rus	33	0.2831
11	ros	9641	0.3315	71	allknn	32	0.5529
12	cc	39	0.3966	72	smote	9457	0.3008
13	adasyn	852	0.7223	73	smote enn	6120	0.3622
14	smote enn	2907	0.3018	74	smote enn	7058	0.3988
15	smote	1056	0.7494	75	smote	558	0.5452
16	allknn	312	0.3739	76	smote tomek	9898	0.3112
17	adasyn	9958	0.8896	78	smote	10254	0.2578
21	allknn	76	0.6779	79	smote enn	4094	0.3123
23	rus	82	0.6639	80	allknn	50	0.2032
24	allknn	249	0.9774	81	ros	266	0.3089
25	allknn	248	0.8743	84	allknn	40	0.3860
26	allknn	248	0.8718	85	allknn	46	0.1972
27	smote enn	2044	0.9775	86	smote	5598	0.0733
28	smote enn	3822	0.2302	88	ros	10222	0.5973
29	smote enn	2746	0.2187	90	ros	10129	0.2854
30	allknn	241	0.1665	95	ros	4260	0.3133
31	allknn	239	0.1571	99	allknn	22	0.3439
32	allknn	220	0.1457	103	ros	783	0.7659
33	adasyn	8683	0.4141	106	adasyn	786	0.3951
34	ros	2314	0.7408	107	smote enn	3985	0.4326
35	smote enn	73	0.5124	108	smote	10467	0.3238
36	ros	4483	0.5289	112	adasyn	8013	0.7299
37	smote enn	3155	0.1140	113	smote	134	0.2916
38	smote	207	0.7813	116	ros	10072	0.8267
39	allknn	55	0.8087	120	smote tomek	1224	0.5667
40	smote tomek	10362	0.5145	123	smote	4240	0.7046
41	smote	277	0.7013	124	rus	17	0.1633
42	smote tomek	4583	0.6752	127	ros	7511	0.4595

43	adasyn	9986	0.6806	128	ros	4240	0.7470
44	ros	2306	0.3316	131	smote	860	0.7923
45	smote enn	5933	0.3907	132	rus	14	0.7867
46	ros	235	0.6649	133	allknn	14	0.3933
47	ros	10587	0.2580	136	ros	583	0.6267
48	rus	123	0.3440	139	adasyn	7776	0.2000
50	allknn	120	0.5071	142	rus	13	0.6267
51	rus	120	0.4265	145	ros	585	0.2203
52	allknn	120	0.2912	153	rus	12	0.8381
53	smote enn	6500	0.1762				

Table C.9: Step 5: sampling algorithms and final samples

Bibliography

- [1] M. Analytics, *The age of analytics: Competing in a data-driven world*, 2016.
- [2] S. Elliot, “Devops and the cost of downtime: Fortune 1000 best practice metrics quantified,” *International Data Corporation (IDC)*, 2014.
- [3] G. Piatetski and W. Frawley, *Knowledge discovery in databases*. MIT press, 1991.
- [4] U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, “Advances in knowledge discovery and data mining,” American Association for Artificial Intelligence, 1996.
- [5] R. Wirth and J. Hipp, “Crisp-dm: Towards a standard process model for data mining,” in *Proceedings of the 4th international conference on the practical applications of knowledge discovery and data mining*, Springer-Verlag London, UK, 2000, pp. 29–39.
- [6] N. Zumel, J. Mount, and J. Porzak, *Practical data science with R*. Manning Shelter Island, NY, 2014.
- [7] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth, “The kdd process for extracting useful knowledge from volumes of data,” *Communications of the ACM*, vol. 39, no. 11, pp. 27–34, 1996.
- [8] A. R. Hevner, S. T. March, J. Park, and S. Ram, “Design science in information systems research,” *MIS quarterly*, pp. 75–105, 2004.
- [9] J. C. Duenas, J. M. Navarro, J. Andion Jiménez, and F. Cuadrado, “Applying event stream processing to network online failure prediction,” *IEEE Communications Magazine*, vol. 56, no. 1, pp. 166–170, 2018.
- [10] J. M. N. González, J. A. Jiménez, and J. C. D. López, “Optimizing failure prediction time windows through genetic algorithms and random forests,” *IEEE Access*, vol. 6, pp. 58 307–58 323, 2018.
- [11] J. M. N. Gonzalez, J. Andion Jimenez, and J. C. D. Lopez, “Root cause analysis of network failures using machine learning and summarization techniques,” *IEEE Communications Magazine*, vol. 55, no. 9, pp. 126–131, 2017.

- [12] S. B. Kotsiantis, I. Zaharakis, and P. Pintelas, "Supervised machine learning: A review of classification techniques," *Emerging artificial intelligence applications in computer engineering*, vol. 160, no. 1, pp. 3–24, 2007.
- [13] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Machine learning*, vol. 3, no. 1, pp. 9–44, 1988.
- [14] S. Muggleton, R. D. King, and M. J. Stenberg, "Protein secondary structure prediction using logic-based machine learning," *Protein Engineering, Design and Selection*, vol. 5, no. 7, pp. 647–657, 1992.
- [15] G. A. Carpenter and S. Grossberg, "A self-organizing neural network for supervised learning, recognition, and prediction," *IEEE Communications Magazine*, vol. 30, no. 9, pp. 38–49, 1992.
- [16] J. H. Chen and S. M. Asch, "Machine learning and prediction in medicine—beyond the peak of inflated expectations," *The New England journal of medicine*, vol. 376, no. 26, p. 2507, 2017.
- [17] S. F. Weng, J. Reps, J. Kai, J. M. Garibaldi, and N. Qureshi, "Can machine-learning improve cardiovascular risk prediction using routine clinical data?" *PloS one*, vol. 12, no. 4, e0174944, 2017.
- [18] S. Athey, "The impact of machine learning on economics," in *The economics of artificial intelligence: An agenda*, University of Chicago Press, 2018, pp. 507–547.
- [19] Q. Kong, D. T. Trugman, Z. E. Ross, M. J. Bianco, B. J. Meade, and P. Gerstoft, "Machine learning in seismology: Turning data into insights," *Seismological Research Letters*, vol. 90, no. 1, pp. 3–14, 2019.
- [20] K. Asim, F. Martinez-Alvarez, A. Basit, and T. Iqbal, "Earthquake magnitude prediction in hindukush region using machine learning techniques," *Natural Hazards*, vol. 85, no. 1, pp. 471–486, 2017.
- [21] R. Sasisekharan, V. Seshadri, and S. M. Weiss, "Data mining and forecasting in large-scale telecommunication networks," *IEEE expert*, vol. 11, no. 1, pp. 37–43, Feb. 1996, ISSN: 08859000. DOI: 10.1109/64.482956.
- [22] V. U. B. Challagulla, F. B. Bastani, I.-L. Yen, and R. A. Paul, "Empirical assessment of machine learning based software defect prediction techniques," *International Journal on Artificial Intelligence Tools*, vol. 17, no. 02, pp. 389–400, 2008.
- [23] Y. Liang, Y. Zhang, M. Jette, A. Sivasubramaniam, and R. Sahoo, "Blue-Gene/L failure analysis and prediction models," in *Proceedings of the International Conference on Dependable Systems and Networks*, vol. 2006, 2006, pp. 425–434, ISBN: 0769526071. DOI: 10.1109/DSN.2006.18.

- [24] R. F. Lima and A. C. M. Pereira, "A fraud detection model based on feature selection and undersampling applied to web payment systems," in *2015 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*, IEEE, vol. 3, 2015, pp. 219–222.
- [25] F. Salfner, M. Lenk, and M. Malek, "A survey of online failure prediction methods," *ACM Computing Surveys*, vol. 42, no. 3, pp. 1–42, 2010, ISSN: 03600300. DOI: 10.1145/1670679.1670680. [Online]. Available: <http://doi.acm.org/10.1145/1670679.1670680>.
- [26] J. Alonso, L. Belanche, and D. R. Avresky, "Predicting software anomalies using machine learning techniques," in *2011 IEEE 10th international symposium on network computing and applications*, IEEE, 2011, pp. 163–170.
- [27] S. Omar, A. Ngadi, and H. H. Jebur, "Machine learning techniques for anomaly detection: An overview," *International Journal of Computer Applications*, vol. 79, no. 2, 2013.
- [28] G. Hamerly and C. Elkan, "Bayesian approaches to failure prediction for disk drives," *Proceedings of the Eighteenth International Conference on Machine Learning*, pp. 202–209, 2001. [Online]. Available: https://www.researchgate.net/publication/2893283%7B%5C_%7DBayesian%7B%5C_%7DApproaches%7B%5C_%7Dto%7B%5C_%7DFailure%7B%5C_%7DPrediction%7B%5C_%7Dfor%7B%5C_%7DDisk%7B%5C_%7DDrives%20http://dl.acm.org/citation.cfm?id=645530.655825.
- [29] A. Pelaez, A. Quiroz, J. C. Browne, E. Chuah, and M. Parashar, "Online failure prediction for hpc resources using decentralized clustering," in *2014 21st International Conference on High Performance Computing (HiPC)*, 2014, pp. 1–9.
- [30] A. Chigurupati, R. Thibaux, and N. Lassar, "Predicting hardware failure using machine learning," in *2016 Annual Reliability and Maintainability Symposium (RAMS)*, 2016, pp. 1–6.
- [31] G. M. Weiss, "Predicting Telecommunication Equipment Failures from Sequences of Network Alarms," Tech. Rep., 2000. [Online]. Available: <https://www.researchgate.net/publication/2628375>.
- [32] M. Shatnawi and M. Hefeeda, "Real-time failure prediction in online services," in *Proceedings - IEEE INFOCOM*, vol. 26, Institute of Electrical and Electronics Engineers Inc., Aug. 2015, pp. 1391–1399, ISBN: 9781479983810. DOI: 10.1109/INFOCOM.2015.7218516.

- [33] G. M. Weiss and H. Hirsh, “Learning to Predict Rare Events in Categorical Time-Series Data,” *AAAI Workshop*, pp. 83–90, 1998. [Online]. Available: www.aaai.org.
- [34] V. Agarwal, C. Bhattacharyya, T. Niranjana, and S. Susarla, “Discovering rules from disk events for predicting hard drive failures,” in *2009 International Conference on Machine Learning and Applications*, IEEE, 2009, pp. 782–786.
- [35] M. Borkowski, W. Fdhila, M. Nardelli, S. Rinderle-Ma, and S. Schulte, “Event-based failure prediction in distributed business processes,” *Information Systems*, vol. 81, pp. 220–235, 2019.
- [36] S. Kotsiantis, D. Kanellopoulos, and P. Pintelas, “Data preprocessing for supervised learning,” *International Journal of Computer Science*, vol. 1, no. 2, pp. 111–117, 2006.
- [37] J. W. Rowe, “A study of the path availability improvements in a large network for various route diversity protection strategies, using failure event simulation,” in *Second IEE National Conference on Telecommunications 1989*, 1989, pp. 167–172.
- [38] M. Bazargan and R. N. McGrath, “Discrete event simulation to improve aircraft availability and maintainability,” in *Annual Reliability and Maintainability Symposium, 2003.*, 2003, pp. 63–67.
- [39] Bowen Hua, Zhaohong Bie, Siu-Kui Au, Wenyuan Li, and Xifan Wang, “Extracting rare failure events in composite system reliability evaluation via subset simulation,” in *2015 IEEE Power Energy Society General Meeting*, 2015, pp. 1–1.
- [40] J. Hebert, “Predicting rare failure events using classification trees on large scale manufacturing data with complex interactions,” in *Proceedings - 2016 IEEE International Conference on Big Data, Big Data 2016*, Institute of Electrical and Electronics Engineers Inc., 2016, pp. 2024–2028, ISBN: 9781467390040. DOI: 10.1109/BigData.2016.7840825.
- [41] R. Vilalta and S. Ma, “Predicting rare events in temporal domains,” in *2002 IEEE International Conference on Data Mining, 2002. Proceedings.*, IEEE, 2002, pp. 474–481.
- [42] T. Akidau, S. Chernyak, and R. Lax, *Streaming systems: the what, where, when, and how of large-scale data processing.* O’Reilly Media, Inc., 2018.
- [43] R. Gwadera, M. J. Atallah, and W. Szpankowski, “Reliable detection of episodes in event sequences,” *Knowledge and Information Systems*, vol. 7, no. 4, pp. 415–437, 2005.

- [44] R. K. Sahoo, A. J. Oliner, I. Rish, M. Gupta, J. E. Moreira, S. Ma, R. Vilalta, and A. Sivasubramaniam, "Critical event prediction for proactive management in large-scale computer clusters," in *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2003, pp. 426–435.
- [45] S. Fu and C.-Z. Xu, "Exploring event correlation for failure prediction in coalitions of clusters," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, 2007, pp. 1–12.
- [46] J. Li, R. J. Stones, G. Wang, X. Liu, Z. Li, and M. Xu, "Hard drive failure prediction using decision trees," *Reliability Engineering & System Safety*, vol. 164, pp. 55–65, 2017.
- [47] E. Chuah, A. Jhumka, S. Narasimhamurthy, J. Hammond, J. C. Browne, and B. Barth, "Linking resource usage anomalies with system failures from cluster log data," in *2013 IEEE 32nd International Symposium on Reliable Distributed Systems*, IEEE, 2013, pp. 111–120.
- [48] E. Fulp, G. Fink, and J. Haack, "Predicting Computer System Failures Using Support Vector Machines," *Proceedings of the First USENIX conference on Analysis of system logs*, pp. 5–5, 2008. [Online]. Available: http://www.rok.informatik.hu-berlin.de/Members/Members/salfner/publications/salfner05predicting.pdf%20http://static.usenix.org/event/was1/tech/full%7B%5C_%7Dpapers/fulp/fulp%7B%5C_%7Dhtml/was108f.html.
- [49] J. Li, X. Ji, Y. Jia, B. Zhu, G. Wang, Z. Li, and X. Liu, "Hard drive failure prediction using classification and regression trees," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, IEEE, 2014, pp. 383–394.
- [50] G. M. Weiss, "Timeweaver: A genetic algorithm for identifying predictive patterns in sequences of events," *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 718–725, 1999. [Online]. Available: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.31.444%7B%5C%7Drep=rep1%7B%5C%7Dtype=pdf>.
- [51] G. M. Weiss and H. Hirsh, "Learning to Predict Extremely Rare Events," Tech. Rep., 2000. [Online]. Available: www.aaai.org.
- [52] W. Yang, D. Hu, Y. Liu, S. Wang, and T. Jiang, "Hard drive failure prediction using big data," in *2015 IEEE 34th Symposium on Reliable Distributed Systems Workshop (SRDSW)*, IEEE, 2015, pp. 13–18.

- [53] H. A. Kholidy, A. Erradi, S. Abdelwahed, A. M. Yousof, and H. A. Ali, “On-line risk assessment and prediction models for autonomic cloud intrusion prevention systems,” in *2014 IEEE/ACS 11th International Conference on Computer Systems and Applications (AICCSA)*, IEEE, 2014, pp. 715–722.
- [54] M. D. Dangut, Z. Skaf, and I. Jennions, “An integrated machine learning model for aircraft components rare failure prognostics with log-based dataset,” *ISA Transactions*, 2020.
- [55] L. Yu, Z. Zheng, Z. Lan, and S. Coghlan, “Practical online failure prediction for blue gene/p: Period-based vs event-driven,” in *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*, IEEE, 2011, pp. 259–264.
- [56] F. Provost, “Machine Learning from Imbalanced Data Sets 101,” Tech. Rep., 2000. [Online]. Available: www.aaai.org.
- [57] G. Haixiang, L. Yijing, J. Shang, G. Mingyun, H. Yuanyue, and G. Bing, “Learning from class-imbalanced data: Review of methods and applications,” *Expert Systems with Applications*, vol. 73, pp. 220–239, 2017.
- [58] C. Beyan and R. Fisher, “Classifying imbalanced data sets using similarity based hierarchical decomposition,” *Pattern Recognition*, vol. 48, no. 5, pp. 1653–1672, 2015.
- [59] M. Khalilia, S. Chakraborty, and M. Popescu, “Predicting disease risks from highly imbalanced data using random forest,” *BMC Medical Informatics and Decision Making*, vol. 11, no. 1, pp. 1–13, Jul. 2011, ISSN: 14726947. DOI: 10.1186/1472-6947-11-51. [Online]. Available: <https://link.springer.com/articles/10.1186/1472-6947-11-51><https://link.springer.com/article/10.1186/1472-6947-11-51>.
- [60] Y. Liu, A. An, and X. Huang, “Boosting prediction accuracy on imbalanced datasets with SVM ensembles,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 3918 LNAI, Springer Verlag, 2006, pp. 107–118, ISBN: 3540332065. DOI: 10.1007/11731139_15. [Online]. Available: https://link.springer.com/chapter/10.1007/11731139_15.
- [61] P. Lingden, A. Alsadoon, P. Prasad, O. H. Alsadoon, R. S. Ali, and V. T. Q. Nguyen, “A novel modified undersampling (mus) technique for software defect prediction,” *Computational Intelligence*, vol. 35, no. 4, pp. 1003–1020, 2019.
- [62] G. M. Weiss, “The Effect of Class Distribution on Classifier Learning: An Empirical Study,” Tech. Rep., 2001. DOI: 10.7282/T3-VPFW-SF95.

- [63] N. Japkowicz, “Why question machine learning evaluation methods,” in *AAAI workshop on evaluation methods for machine learning*, 2006, pp. 6–11.
- [64] P. Branco, L. Torgo, and R. Ribeiro, “A survey of predictive modelling under imbalanced distributions,” *arXiv preprint arXiv:1505.01658*, 2015.
- [65] O. Loyola-Gonzalez, J. F. Martinez-Trinidad, J. A. Carrasco-Ochoa, and M. Garcia-Borroto, “Study of the impact of resampling methods for contrast pattern based classifiers in imbalanced databases,” *Neurocomputing*, vol. 175, pp. 935–947, 2016.
- [66] J. Davis and M. Goadrich, “The relationship between precision-recall and roc curves,” in *Proceedings of the 23rd international conference on Machine learning*, 2006, pp. 233–240.
- [67] G. M. Weiss and H. Hirsh, “Learning to predict rare events in event sequences,” in *KDD*, vol. 98, 1998, pp. 359–363.
- [68] S. Zhang, W. Meng, J. Bu, S. Yang, Y. Liu, D. Pei, J. Xu, Y. Chen, H. Dong, X. Qu, and L. Song, “Syslog processing for switch failure diagnosis and prediction in datacenter networks,” in *2017 IEEE/ACM 25th International Symposium on Quality of Service, IWQoS 2017*, Institute of Electrical and Electronics Engineers Inc., Jul. 2017, ISBN: 9781509019830. DOI: 10.1109/IWQoS.2017.7969130.
- [69] V. H. Shah, “Machine learning techniques for stock prediction,” *Foundations of Machine Learning—Spring*, vol. 1, no. 1, pp. 6–12, 2007.
- [70] S. Shen, H. Jiang, and T. Zhang, “Stock market forecasting using machine learning algorithms,” *Department of Electrical Engineering, Stanford University, Stanford, CA*, pp. 1–5, 2012.
- [71] T. Suchatpong and K. Bhumkittipich, “Hard disk drive failure mode prediction based on industrial standard using decision tree learning,” in *2014 11th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*, 2014, pp. 1–4.
- [72] S. Jhaveri, I. Khedkar, Y. Kantharia, and S. Jaswal, “Success prediction using random forest, catboost, xgboost and adaboost for kickstarter campaigns,” in *2019 3rd International Conference on Computing Methodologies and Communication (ICCMC)*, 2019, pp. 1170–1173.
- [73] Z. Cai, S. Sun, S. Si, and N. Wang, “Modelling of failure prediction bayesian network based on fault tree analysis,” in *2010 IEEE 17Th International Conference on Industrial Engineering and Engineering Management*, 2010, pp. 937–941.

- [74] J. Yin, F. Dong, and N. Wang, "Modified gram-schmidt algorithm for extreme learning machine," in *2009 Second International Symposium on Computational Intelligence and Design*, vol. 2, 2009, pp. 517–520.
- [75] G. Zhang, "A modified svm classifier based on rs in medical disease prediction," in *2009 Second International Symposium on Computational Intelligence and Design*, vol. 1, 2009, pp. 144–147.
- [76] Y. Ma and D. Zhai, "A modified multi-output support vector regression machine based on data dependent kernel function," in *2009 International Joint Conference on Computational Sciences and Optimization*, vol. 2, 2009, pp. 939–941.
- [77] P. Chandrasekar, K. Qian, H. Shahriar, and P. Bhattacharya, "Improving the prediction accuracy of decision tree mining with data preprocessing," in *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, vol. 2, 2017, pp. 481–484.
- [78] W. Liu, S. Liu, Q. Gu, J. Chen, X. Chen, and D. Chen, "Empirical studies of a two-stage data preprocessing approach for software fault prediction," *IEEE Transactions on Reliability*, vol. 65, no. 1, pp. 38–53, 2016.
- [79] S. Lo, "The effects of feature selection and model selection on the correctness of classification," in *2010 IEEE International Conference on Industrial Engineering and Engineering Management*, 2010, pp. 989–993.
- [80] A. M. Pirbazari, A. Chakravorty, and C. Rong, "Evaluating feature selection methods for short-term load forecasting," in *2019 IEEE International Conference on Big Data and Smart Computing (BigComp)*, 2019, pp. 1–8.
- [81] A. Das, F. Mueller, and B. Rountree, "Aarohi: Making real-time node failure prediction feasible," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, pp. 1092–1101.
- [82] T. Ghosh, D. Sarkar, T. Sharma, A. Desai, and R. Bali, "Real time failure prediction of load balancers and firewalls," in *2016 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, 2016, pp. 822–827.
- [83] Y. Liang, Y. Zhang, H. Xiong, and R. Sahoo, "Failure prediction in IBM BlueGene/L event logs," in *Proceedings - IEEE International Conference on Data Mining, ICDM*, 2007, pp. 583–588, ISBN: 0769530184. DOI: 10.1109/ICDM.2007.46.

- [84] Z. Zheng, Z. Lan, R. Gupta, S. Coghlan, and P. Beckman, "A practical failure prediction with location and lead time for blue gene/p," in *2010 International Conference on Dependable Systems and Networks Workshops (DSN-W)*, IEEE, 2010, pp. 15–22.
- [85] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic minority over-sampling technique," *Journal of Artificial Intelligence Research*, vol. 16, pp. 321–357, Jan. 2002, ISSN: 10769757. DOI: 10.1613/jair.953. arXiv: 1106.1813.
- [86] R. Pears, J. Finlay, and A. M. Connor, "Synthetic minority over-sampling technique (smote) for predicting software build outcomes," *arXiv preprint arXiv:1407.2330*, 2014.
- [87] I. Irrera and M. Vieira, "A practical approach for generating failure data for assessing and comparing failure prediction algorithms," in *2014 IEEE 20th Pacific Rim International Symposium on Dependable Computing*, IEEE, 2014, pp. 86–95.
- [88] J. A. Duraes and H. S. Madeira, "Emulation of software faults: A field data study and a practical approach," *Ieee transactions on software engineering*, vol. 32, no. 11, pp. 849–867, 2006.
- [89] J. Misra, "Distributed discrete-event simulation," *ACM Computing Surveys (CSUR)*, vol. 18, no. 1, pp. 39–65, 1986.
- [90] R. M. Fujimoto, "Parallel discrete event simulation," *Communications of the ACM*, vol. 33, no. 10, pp. 30–53, 1990.
- [91] W. R. Franta and K. Maly, "An efficient data structure for the simulation event set," *Communications of the ACM*, vol. 20, no. 8, pp. 596–602, 1977.
- [92] P.-O. Siebers, C. M. Macal, J. Garnett, D. Buxton, and M. Pidd, "Discrete-event simulation is dead, long live agent-based simulation!" *Journal of Simulation*, vol. 4, no. 3, pp. 204–210, 2010.
- [93] A. S. Koulinitch and L. B. Sheremetov, "Coordination and communication issues in multi-agent expert system: Concurrent configuration design advisor," *Expert Systems with Applications*, vol. 15, no. 3-4, pp. 295–307, 1998.
- [94] O. Arsene, I. Dumitrache, and I. Mihu, "Expert system for medicine diagnosis using software agents," *Expert Systems with Applications*, vol. 42, no. 4, pp. 1825–1834, 2015.
- [95] C. M. Macal and M. J. North, "Tutorial on agent-based modeling and simulation," in *Proceedings of the Winter Simulation Conference, 2005.*, IEEE, 2005, 14–pp.

- [96] R. J. Allan, *Survey of agent based modelling and simulation tools*. Science & Technology Facilities Council London, 2010.
- [97] C. Gaber, B. Hemery, M. Achemlal, M. Pasquet, and P. Urien, "Synthetic logs generator for fraud detection in mobile transfer services," in *2013 International Conference on Collaboration Technologies and Systems (CTS)*, IEEE, 2013, pp. 174–179.
- [98] H. Kavak, J. J. Padilla, C. J. Lynch, and S. Y. Diallo, "Big data, agents, and machine learning: Towards a data-driven agent-based modeling approach," in *Proceedings of the Annual Simulation Symposium*, 2018, pp. 1–12.
- [99] A. W.-U. Ashraf, M. Budka, and K. Musial, "Simulation and augmentation of social networks for building deep learning models," *arXiv preprint arXiv:1905.09087*, 2019.
- [100] E.-W. Augustijn, S. A. Abdulkareem, M. H. Sadiq, and A. A. Albabawat, "Machine learning to derive complex behaviour in agent-based modelling," in *2020 International Conference on Computer Science and Software Engineering (CSASE)*, IEEE, 2020, pp. 284–289.
- [101] D. B. Fuller, E. F. de Arruda, and V. J. M. Ferreira Filho, "Learning-agent-based simulation for queue network systems," *Journal of the Operational Research Society*, pp. 1–17, 2019.
- [102] E. Lundin, H. Kvarnstrom, and E. Jonsson, "A synthetic fraud data generation methodology," in *International Conference on Information and Communications Security*, Springer, 2002, pp. 265–277.
- [103] M. Pereda, J. I. Santos, and J. M. Galan, "A brief introduction to the use of machine learning techniques in the analysis of agent-based models," in *Advances in Management Engineering*, Springer, 2017, pp. 179–186.
- [104] O. Batata, V. Augusto, and X. Xie, "Mixed machine learning and agent-based simulation for respite care evaluation," in *2018 Winter Simulation Conference (WSC)*, IEEE, 2018, pp. 2668–2679.
- [105] D. Pyle, *Data preparation for data mining*. morgan kaufmann, 1999.
- [106] F. Salfner, "Predicting failures with hidden Markov models," *Proceedings of 5th European Dependable Computing Conference (EDCC-5)*, pp. 41–46, 2005. [Online]. Available: <http://www.rok.informatik.hu-berlin.de/Members/Members/salfner/publications/salfner05predicting.pdf>.
- [107] Y. Watanabe, H. Otsuka, M. Sonoda, S. Kikuchi, and Y. Matsumoto, "Online failure prediction in cloud datacenters by real-time message pattern learning," in *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*, IEEE, 2012, pp. 504–511.

- [108] R. Baldoni, L. Montanari, and M. Rizzuto, “On-line failure prediction in safety-critical systems,” *Future Generation Computer Systems*, vol. 45, pp. 123–132, 2015.
- [109] S. Zhang, Y. Liu, W. Meng, Z. Luo, J. Bu, S. Yang, P. Liang, D. Pei, J. Xu, Y. Zhang, *et al.*, “Prefix: Switch failure prediction in datacenter networks,” *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 2, no. 1, pp. 1–29, 2018.
- [110] B. W. Matthews, “Comparison of the predicted and observed secondary structure of t4 phage lysozyme,” *Biochimica et Biophysica Acta (BBA)-Protein Structure*, vol. 405, no. 2, pp. 442–451, 1975.
- [111] J. Bergstra, J. B. Ca, and Y. B. Ca, “Random Search for Hyper-Parameter Optimization Yoshua Bengio,” Tech. Rep., 2012, pp. 281–305. [Online]. Available: <http://scikit-learn.sourceforge.net..>
- [112] H. He, Y. Bai, E. A. Garcia, and S. Li, “ADASYN: Adaptive synthetic sampling approach for imbalanced learning,” in *Proceedings of the International Joint Conference on Neural Networks*, 2008, pp. 1322–1328, ISBN: 9781424418213. DOI: 10.1109/IJCNN.2008.4633969.
- [113] H. Li, *Smile*, <https://haifengl.github.io>, 2014.
- [114] K. El Emam, L. Mosquera, and R. Hoptroff, *Practical Synthetic Data Generation: Balancing Privacy and the Broad Availability of Data*. O’Reilly Media, 2020.
- [115] J. A. J. M. N. J. C. D. Lopez, *Computer network events*, 2019. DOI: 10.21227/dbgv-zz92. [Online]. Available: <http://dx.doi.org/10.21227/dbgv-zz92>.
- [116] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger, *The scala language specification*, 2004.
- [117] M. Odersky, L. Spoon, and B. Venners, *Programming in scala*. Artima Inc, 2008.
- [118] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica, *et al.*, “Spark: Cluster computing with working sets,” *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.
- [119] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, *et al.*, “Spark sql: Relational data processing in spark,” in *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, 2015, pp. 1383–1394.

- [120] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia, “Structured streaming: A declarative api for real-time applications in apache spark,” in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 601–613.
- [121] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, *et al.*, “Mllib: Machine learning in apache spark,” *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 1235–1241, 2016.
- [122] T. pandas development team, *Pandas-dev/pandas: Pandas*, version latest, Feb. 2020. DOI: 10.5281/zenodo.3509134. [Online]. Available: <https://doi.org/10.5281/zenodo.3509134>.
- [123] W. McKinney, “Data Structures for Statistical Computing in Python,” in *Proceedings of the 9th Python in Science Conference*, S. van der Walt and J. Millman, Eds., 2010, pp. 56–61. DOI: 10.25080/Majora-92bf1922-00a.
- [124] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, “Scikit-learn: Machine learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.
- [125] S. Wold, K. Esbensen, and P. Geladi, “Principal component analysis,” *Chemometrics and intelligent laboratory systems*, vol. 2, no. 1-3, pp. 37–52, 1987.
- [126] D. M. Blei, A. Y. Ng, and M. I. Jordan, “Latent dirichlet allocation,” *Journal of machine Learning research*, vol. 3, no. Jan, pp. 993–1022, 2003.
- [127] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, “The hadoop distributed file system,” in *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, Ieee, 2010, pp. 1–10.
- [128] J. Subramanian and R. Simon, “Overfitting in prediction models—is it a problem only in high dimensions?” *Contemporary clinical trials*, vol. 36, no. 2, pp. 636–641, 2013.
- [129] J. D. Rodriguez, A. Perez, and J. A. Lozano, “Sensitivity analysis of k-fold cross validation in prediction error estimation,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 32, no. 3, pp. 569–575, 2010.
- [130] J. A. Nelder and R. W. M. Wedderburn, “Generalized linear models,” *Journal of the Royal Statistical Society. Series A (General)*, vol. 135, no. 3, pp. 370–384, 1972, ISSN: 00359238. [Online]. Available: <http://www.jstor.org/stable/2344614>.

- [131] E. Alpaydin, *Introduction to machine learning*. MIT press, 2014.
- [132] B. E. Boser, I. M. Guyon, and V. N. Vapnik, “A training algorithm for optimal margin classifiers,” in *Proceedings of the fifth annual workshop on Computational learning theory*, 1992, pp. 144–152.
- [133] Ö. F. Arar and K. Ayan, “A feature dependent naive bayes approach and its application to the software defect prediction problem,” *Applied Soft Computing*, vol. 59, pp. 197–209, 2017.
- [134] X. Feng, S. Li, C. Yuan, P. Zeng, and Y. Sun, “Prediction of slope stability using naive bayes classifier,” *KSCE Journal of Civil Engineering*, vol. 22, no. 3, pp. 941–950, 2018.
- [135] M. Granik and V. Mesyura, “Fake news detection using naive bayes classifier,” in *2017 IEEE First Ukraine Conference on Electrical and Computer Engineering (UKRCON)*, IEEE, 2017, pp. 900–903.
- [136] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen, *Classification and regression trees*. CRC press, 1984.
- [137] L. Breiman, “Random forests,” *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.
- [138] J. H. Friedman, “Stochastic gradient boosting,” *Computational statistics & data analysis*, vol. 38, no. 4, pp. 367–378, 2002.
- [139] R. E. Schapire, “The strength of weak learnability,” *Machine learning*, vol. 5, no. 2, pp. 197–227, 1990.
- [140] S. Węglarczyk, “Kernel density estimation and its application,” *ITM Web Conf.*, vol. 23, p. 00 037, 2018. DOI: 10.1051/itmconf/20182300037. [Online]. Available: <https://doi.org/10.1051/itmconf/20182300037>.
- [141] H. S. Nwana *et al.*, “Software agents: An overview,” *Knowledge engineering review*, vol. 11, no. 3, pp. 205–244, 1996.
- [142] C. Hewitt, P. Bishop, and R. Steiger, “Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence,” in *Advance Papers of the Conference*, Stanford Research Institute, vol. 3, 1973, p. 235.