POLITECNICO DI TORINO

Master's Degree in Mathematical Engineering

Master's Degree Thesis

Fault detection and explanation applied to engine control unit data

Supervisor
prof. Francesco VACCARINO

Candidate
Francesco SARACCO

Company Tutor
Emanuele GALLO

Academic Year 2020-2021

Summary

Anomaly detection is a major field of research in Machine Learning and Artificial Intelligence literature, particularly in industrial applications. The effective and precocious identification of issues in an industrial process is critical in order to operate immediate interventions, avoid damages and thus reduce costs. However, an issue that is often overlooked in the context of anomaly detection is that of explaining anomalies, thus sacrificing interpretability. The anomaly detection system thus becomes a so-called 'black box', which can only be trusted blindly. This thesis, hosted by Data Reply, presents a ML-based algorithm aimed at the extraction of human-comprehensible rules which characterise and explain the anomalous class in the context of a supervised dataset with binary labels, obtained by a previous phase of anomaly detection. It may be regarded as a tool to extend the latter with an *explanation* layer, providing a greater interpretability of the detection of anomalies and explaining which features distinguish them from normal data points. In particular, a Random Forest is fitted on the labelled dataset, in order to learn the salient characteristics of the anomalous class. Then, a heuristic search procedure is employed to select, from the set of all the rules the RF is composed of, a subset which optimises a suitable performance measure. This last step allows to reduce the total number of rules to a manageable level, while at the same time retaining as much information as possible. The proposed solution was applied to an automotive case on data gathered from engine control units on industrial vehicles for an important multinational client firm, leader in the development, production, sales and service of power trains for on-road, off-road, marine and power generation applications. The objective was to obtain, from data describing vehicles operations, a characterisation of the system's fault conditions, both effective in detecting issues and comprehensible by the client firm's professionals. This would allow to further investigate anomalies exploiting these rules as a starting point for tests and on field operations. The algorithm was tested on a fleet of vehicles which had known issues, in order to validate the results obtained through the judgement and domain knowledge of the client firm's professionals. The results obtained have been found satisfactory by the latter and a follow up project has received funding from the client company.

Acknowledgements

I would first like to thank my supervisor, Professor Francesco Vaccarino, whose suggestions offered me the right starting point and whose valuable advice helped me to put my work on a more theoretically sound foundation, greatly enhancing its content.

I would like to acknowledge my colleagues from my internship at Data Reply for including me from day one. In particular, I would like to thank Emanuele, for the valuable experience and learning opportunity he offered me, and Valerio, who followed me tirelessly.

Finally, I would like to thank my family (both my original family and my in-laws), especially my parents who encouraged me to embark on this path, and Rossella, who believes in me and supports me every single day.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Machine Learning (ML) (or *Statistical Learning*) is a very broad term referring to ideas and techniques systematically developed from the last decades of the XX century, but whose roots trace back to much earlier. It can be placed at the intersection of various knowledge fields, primarily Statistics, Optimization and Informatics. The common high-level objective of all these techniques is the automatic learning and recognition of patterns in data, which in turn may be exploited in several ways. In general, they may provide insights about the data generating process and generate models that can be used for predictive purposes.

ML has had a huge impact on various areas and Industry as a whole is no exception. It has improved processes, reduced errors and waste and unlocked capabilities that were unimaginable even just a decade ago.

This thesis presents a project coordinated by Data Reply for an important multinational company, leader in the development, production, sales and service of engines for on-road, off-road, marine and power generation applications. For some time now, the client company has been equipping its engines with telematic control units that continuously collect data on the status of the monitored vehicles and is now starting R&D projects, like the one we describe here, to implement ML-based algorithms and pipelines to extract valuable

insights about their products. In the context of this work, we implemented an algorithm aimed at the extraction of human-comprehensible rules which characterise the anomalous class in the context of a supervised dataset with binary labels, obtained by an anomaly detection phase. This tool is meant to add an explanation layer to a pipeline aimed at the identification of anomalies in a set of data, in order to enhance the comprehension of the nature of the anomalies themselves.

The latter was applied, for the client company, on data gathered from telematics systems on industrial vehicles. The objective was to obtain, from data describing vehicles operations, a characterisation of the system's fault conditions, which could be comprehensible by the client firm's professionals. This would allow to further investigate anomalies, exploiting these rules as a starting point for tests and on field operations. Given the application domain, please note that we will use the terms "fault" and "anomaly" somewhat interchangeably, even though, in general, they are not synonyms. The assumption, in this case, is that a fault on the engine produces an anomaly in the data which describe the vehicle's operations.

In this introductory chapter, we briefly describe the generation and collection process of data in 1.1 and the task considered for this project in 1.2.

1.1 The fault monitoring system

As we anticipated above, Data Reply developed for the client company a cloud-based system which is in charge of ingesting, processing and organizing all data flows coming from the vehicles equipped with on-board telematics boxes. These devices constantly send information coming from sensors and radars of the vehicle and enable the manufacturer to know the status of its products almost in real-time, permitting a constant monitoring (in particular, for what concerns faults) and an efficient customer service. We will briefly describe the fault monitoring system, while referring the reader to [4] for more details on its development and components.

1.1.1 Data ingestion

Raw data coming from telematics boxes are sent to a dispatcher, that is in charge of distinguishing the different data flows and forwarding them to the related queue, for parsing and processing, before they are stored in SQL tables. The data flows are:

- **Engine-on**: data packages sent when the vehicle is switched on, containing geographical infos and system-related parameters values.
- **Engine**: data packages sent every 15 minutes when the engine is operating, until it is switched off. They contain data similar to those contained in engine-on packages, providing a continuous monitoring of the vehicle.
- **Faults**: data packages sent on event, when the telematics detect a new fault. They contain geographical infos, id(s) of fault(s) and the total number of occurrences. Note that, in general, faults signals are continuous in time, meaning that usually a fault turns on, remains active for a certain period of time and eventually turns off. Also, faults are categorized in severity levels (Low, Medium and High).
- **Vehicle Anagraphic Data**: data packages sent only once, at the box installation. They contain anagraphic data about vehicle and user and were not considered in our project.

After suitable preprocessing, all data flows are stored in the related storage solution. These data are the input for the core application of the system, described in the following section.

1.1.2 The rule engine

This component is the main part of the whole fault monitoring system. It is a program written in Scala that takes as input the data flows described before along with predefined rules and outputs information related to the status of the vehicle, detecting issues and

faults on the engine. This component is necessary because, as we will explain later, the faults detected by the engine control unit are not very accurate. If this signal were used directly, it would produce a high number of false positives. For this reason, rules are defined (based on the system parameters, the types of faults detected and the number of occurrences) which, if triggered, signal a certain problem. Rules can be *simple* or *complex*: simple rules are sets of conditions based on vehicle data concatenated with AND and OR expressions, while complex rules are sets of simple rules.
Our project is based on these rules and the process to generate them. Currently, this procedure is carried out as follows. Each rule is:

- manually designed by automotive engineers
- programmed in suitable language for the rule engine
- tested on development environments
- tested on a predefined set of validation vehicles
- deployed on production

This process is evidently time-consuming, error prone and also often requires manual tests to reproduce fault conditions, which are difficult to conduct and typically costly. Our project proposes a first ML-based solution to automatically extract rules directly from on-board data coming from vehicle boxes.

1.1.3 Web interface

The results of the processing steps described before are presented to the user by means of a web-based interface, from which the client company's professionals and customer care can monitor all vehicles in real-time and have all sort of details about them.

1.2 Task description

As mentioned before, the main part of our project was devoted to the implementation of an algorithm (originally presented in [5]) aimed at extracting from a Random Forest Classifier a minimal set of rules, which exhibits similar classification performances. These rules may be used to interpret the model's classification process, obtain insights about the categories considered and become the input of further statistical analysis. Actually, our implementation presents some modifications with respect to the original algorithm, which were aimed at repurposing the algorithm to a different objective, namely, the characterisation of anomalies (found by an anomaly detection algorithm) in terms of rules based on the system's parameters, just like the rules employed by the rule engine. Furthermore, we parallelized the whole computation, dramatically improving the processing time.

Chapter 2

Preprocessing and Exploratory Analysis

A thorough analysis of data is a fundamental step of any workflow aimed at the application of ML techniques. The specific form and features of the data at hand should lead to the most appropriate algorithm to exploit, given the task to accomplish. All the more so in cases such as ours, where there was not a ready-made dataset and we had to build it from the data stored in the client company's Data Lake. In this chapter, we will address this important task. In section 2.1, we present the data in their raw form, as it is stored after ingestion. Then, section 2.2 is devoted to the preprocessing applied, while section 2.3 discusses the labelling process. Finally, some standard exploratory analysis will allow to get some first insights in section 2.4.

2.1 Raw data

As mentioned in the Introduction, after suitable automatic preprocessing, data coming from the vehicles boxes is stored in a Data Warehouse, in the form of structured tables. The dataset exploited in our project was made using information from four main

tables: *can_signal*, *can_header*, *dm1_dtc* and *dm1_header*. The *can* prefix refers to data packages containing the values of all system parameters monitored, while the *dm1* prefix identifies data packages containing information about faults detected by the telematics box. All these tables will be analysed in the following sections. In general, a *can* or *dm1* package is identified by a unique id (that is related to a unique timestamp) and its content is divided among the signal table, containing parameters values, and the header table, which contains anagraphical and geographical information about the vehicle. The id column allows to recompose the complete package.

2.1.1 can_signal Table

The *can_signal* table contains the values of all the system parameters that are monitored by the telematics box. A *can* package is always sent at the vehicle switch on. From that instant on, packages are sent periodically every 15 minutes and a last one is sent at the vehicle switch off.

	uuid	parameter_id	parameter_id_hex	min_value	max_value	average_value	standard_deviation_value	first_value	last_value
1	ea417d5b-8491-4bcb-8b35-c474d7f41ead	1093	445	null	null	0	null	null	null
2	ea417d5b-8491-4bcb-8b35-c474d7f41ead	1092	444	null	null	0	null	null	null
3	ea417d5b-8491-4bcb-8b35-c474d7f41ead	1089	441	null	null	0	null	null	null
4	ea417d5b-8491-4bcb-8b35-c474d7f41ead	1067	42B	null	null	0	null	null	null
5	ea417d5b-8491-4bcb-8b35-c474d7f41ead	1052	41C	null	null	null	null	null	0
6	ea417d5b-8491-4bcb-8b35-c474d7f41ead	1008	3F0	null	null	null	null	null	12
7	ea417d5b-8491-4bcb-8b35-c474d7f41ead	1235	4D3	null	null	54.484375	null	null	null

Figure 2.1. A snapshot of the can_signal table

Each row of the table contains the following features:

- **uuid**: unique identifier of the *can* package
- **parameter_id**, **parameter_id_hex**: decimal and hexadecimal identifier of the system parameter
- **min_value**, **max_value**, **average_value**, **standard_deviation_value**, **first_value**, **last_value**, **integration_value**, **instant_value**: depending on the specific parameter, only a subset of these features has a non-Null value. Actually, in most cases

only one feature for each parameter has non-Null values. For example, some parameters are monitored by their mean value in the 15 minutes interval, other parameters by their maximum value and so on. Note that the vehicle's parameters are either numerical or categorical.

- **insert_date**: the timestamp of the instant of insertion of the package in the table
- **year**, **month**: year and month value extracted by the previous column

As we mentioned above, a package emitted at a specific timestamp contains the values of all parameters being monitored by the vehicle telematics. Thus, each package is split in the table between multiple rows, as can be seen in figure 2.1.

2.1.2 can_header Table

The *can_header* table contains information that identify the specific vehicle and its telematics box, along with data about the position of the vehicle at the instant of emission of the package.

	uuid	gateway_serial_number	dongle_serial_number	payload_type	device_type	message_type	message_version	engine_hours
1	a2c61e94-61e0-4c43-bb50-dbc2dedfdabf	TC1MT000015	TOYMT000137	el	TGWFI	1	0	2214
2	93544d1b-6b7a-41a3-971f-d10bccb38163	TC1KT000046	TOMKT000009	el	TGWFI	1	0	1949
3	4189fe15-48e2-4aef-9fd3-e0eac9bf3db1	TC1KT000065	TOMKT000067	el	TGWFI	1	0	3247
4	714a3662-c15e-486c-956c-b8952b624046	TC1MT000022	TOYMT000163	st	TGWFI	0	0	0
5	0b163183-bfb4-45b7-aea4-cf490e4fff49	TC1MT000041	TOYMT000170	el	TGWFI	1	0	2328
6	dbb899b3-f6cd-466f-aa95-2b8fb066c08f	TC1LT000021	TOMLT000350	st	TGWFI	0	0	876
7	aa78eca1-888f-4f6e-8646-25f105f07aec	TC1KT000039	TOMLT000491	el	TGWFI	1	0	0

Figure 2.2. A snapshot of the can_header table

Each row of the table contains the following features:

- **uuid**: : unique identifier of the *can* package
- **gateway_serial_number**: serial number of the vehicle's gateway box
- **dongle_serial_number**: serial number of the vehicle telematics box

- **payload_type**: identifier of data flow
- **device_type**: type of transmitting device
- **message_type**: categorical value representing the data flow of the package (0 : Engine-on, 1 : Engine, 2 : Engine-off)
- **message_version**: version of the message
- **engine_hours**: total operating time of the vehicle, expressed in hours
- **device_timestamp**: timestamp of the data package
- **meters_above_sea_level**: information about geographical position of the vehicle
- **speed**, **angle**: instantaneous speed and current wheel angle of the vehicle
- **position_device_timestamp**: timestamp of position acquisition
- **geo_type**: GPS mode
- **longitude**, **latitude**: information about geographical position of the vehicle
- **gps_signal_quality**: quality of GPS signal
- **id_trip**: counter of operating periods (increments at switch on)
- **obs_period**: time window for computing aggregate values
- **is_key_on**: indicator of key insertion
- **timestamp_acquisition**: timestamp of acquisition of package from data dispatcher
- **insert_date**: the timestamp of the instant of insertion of the package in the table
- **year**, **month**: year and month value extracted by the previous column

Note that, unlike the previous table, in this case there is only one row for each data package. A snapshot of the table is shown in figure 2.2.

2.1.3 dm1_dtc Table

The *dm1_dtc* table contains specific information about faults.

	uuid	dtc_node	dtc_spn	dtc_spn_hex	dtc_fmi	dtc_fmi_hex	dtc_oc	insert_date	year	month
1	83075f7a-4d68-4cb4-aba3-1359b790a7c8	0	1485	5CD	13	D	12	2021-10-01T07:47:02.050+0000	2021	10
2	3c915e25-df24-4cc9-bdc1-423a35e7da13	0	17426	4412	4	4	1	2021-10-01T07:22:13.937+0000	2021	10
3	c70346f4-3c8c-4cc1-b054-4a2a7daefedc	0	1382	566	31	1F	126	2021-10-01T06:42:16.100+0000	2021	10
4	c70346f4-3c8c-4cc1-b054-4a2a7daefedc	0	520797	7F25D	18	12	1	2021-10-01T06:42:16.097+0000	2021	10
5	bb28bf8a-00eb-4dc3-8a23-1a4e753f3a40	0	520797	7F25D	18	12	1	2021-10-01T06:34:20.170+0000	2021	10
6	3049aa6b-9b9e-4faa-a3f9-9d51ab5e64ca	0	3056	BF0	2	2	1	2021-10-01T06:26:40.223+0000	2021	10
7	3049aa6b-9b9e-4faa-a3f9-9d51ab5e64ca	0	516178	7E052	31	1F	1	2021-10-01T06:26:40.133+0000	2021	10

Figure 2.3. A snapshot of the dm1_dtc table

Each row of the table contains the following features:

- **uuid**: : unique identifier of the *can* package
- **dtc_node**: identifier of component affected by fault
- **dtc_spn**, **dtc_fmi**: decimal *spn* and *fmi* codes. A (*spn*,*fmi*) couple identifies a specific fault
- **dtc_spn_hex**, **dtc_fmi_hex**: hexadecimal *spn* and *fmi* codes
- **dtc_oc**: total number of occurrences of the fault
- **insert_date**: the timestamp of the instant of insertion of the package in the table
- **year**, **month**: year and month value extracted by the previous column

Also in this case, if a *dm1* package contains multiple faults, it is split on multiple rows in the table, as figure 2.3 shows.

2.1.4 dm1_header Table

The *dm1_header* table contains the same columns as the *can_header* table, with their information obviously pertaining the *dm1* packages. The features have the exact same meaning, thus we will not list them again.

2.2 Preprocessing

The client company selected a fleet of five identical vehicles suitable for the development of the project. These vehicles have well-known criticalities and are therefore an interesting case study. In addition, through the expertise of Data Reply professionals about the system that generates the data, some of the features presented above were disregarded a priori, given that they are not considered reliable sources of information.

2.2.1 Steps description

Given this considerations, we will now describe the preprocessing steps applied to the raw data. This phase was conducted through Apache Spark framework, in its Python implementation (PySpark), on Azure Databricks, a cloud platform that provides automated cluster management and IPython-style notebooks.
The preprocessing applied consisted in the following steps:

1. The *can_signal* and *can_header* tables were joined on the value of **uuid** feature, as were the *dm1_dtc* and *dm1_header* tables, in order to obtain tables with the complete information about each package

2. Old test data and unreliable features were filtered out, as discussed above

3. The multiple columns relating to the parameter's value were merged in one single **value** column. In cases in which a parameter is monitored in more than one mode, the content of the **value** column was obtained by one of the old columns with the following priority order: **average_value**, **min_value**, **max_value**, **first_value**, **last_value**, **instant_value**, **integration_value**

4. To gather the content of each *can* package on one single line, the distinct values of **parameter_id** were extracted. Then, a new table schema was built, using

the features **uuid**, **gateway_serial_number**, **message_type**, **id_trip** and **device_timestamp** along with the distinct parameters extracted. The table described by this schema was built from the previous one through a pivot operation

5. The distinction between numerical and categorical features was made in one of two ways: in some cases the information could be extracted a priori from other tables; in all other cases, all parameters with no more than 15 distinct unique values were considered categorical, while the remaining ones were considered numerical. This maximum number of distinct unique values was chosen empirically, through a study of the data itself

6. Some features had residual Null values, that were forward-filled, motivating this procedure on the way in which the system updates its status: in fact, a parameter value is transmitted through a *can* package only if its value has changed since the last package emitted. Thus, a Null value just means that the corresponding parameter has not changed its value, so the forward-filling procedure is justified

2.2.2 Final dataset construction

The steps described above were applied individually to each vehicle of the fleet, obtaining five datasets. The final dataset was then produced by concatenating them, thus disregarding the identity of the vehicles. We recall that this passage is allowed by the fact that the vehicles selected are identical. Despite this, a couple of parameters (related to specific functionalities of some engines) were not shared by all vehicles of the fleet and were thus discarded. The selected vehicles operate daily, so the dataset is periodically rebuilt and grows in size.

From this table, the features table used in the algorithms applied (the table typically named X) was extracted keeping all parameters features along with **message_type** categorical feature.

2.3 Labels design

As the careful reader may have already noticed, the dataset as described until now does not contain any information from the fault-related tables. In fact, those tables were used in the labelling step, in order to obtain a supervised dataset.
Clearly, there is not a unique way to obtain labels from the information contained in the *dm1* tables, even once the learning task is determined, nor to attribute those labels to the samples and the choices that this kind of situations request are always disputable. In fact, we designed and tested two different labelling procedures, which we discuss in the following sections.

2.3.1 Labelling logics: closest configuration

Given that the *can* data packages are separated by time intervals of non-negligible duration, a specific SQL query was designed to match each *dm1* package, which, we recall, signals one or more faults detected on the vehicle, with the *can* package closest in time, after or before it. The objective was to relate a fault signal to the information about the system configuration nearest in time available in the data. As for the labels, each *can* package that was matched to a *dm1* package was labelled with the value 1, that indicates a fault, while the remaining samples were labelled with 0, meaning the absence of faults. A summary diagram of the process described is shown in figure 2.4: in the case represented, the first and last *can* packages are labelled with 0, while the middle one is labelled with 1.

As we already discussed, other choices would have been possible. For example, we could have restricted a *dm1* package to be matched to the closest *can* package emitted before (or after) it.
The labelling process described produces the distribution shown in figure 2.5.

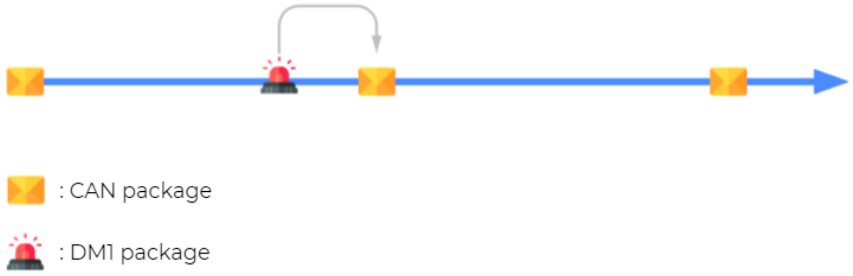

Figure 2.4. A schematic representation of the labelling process

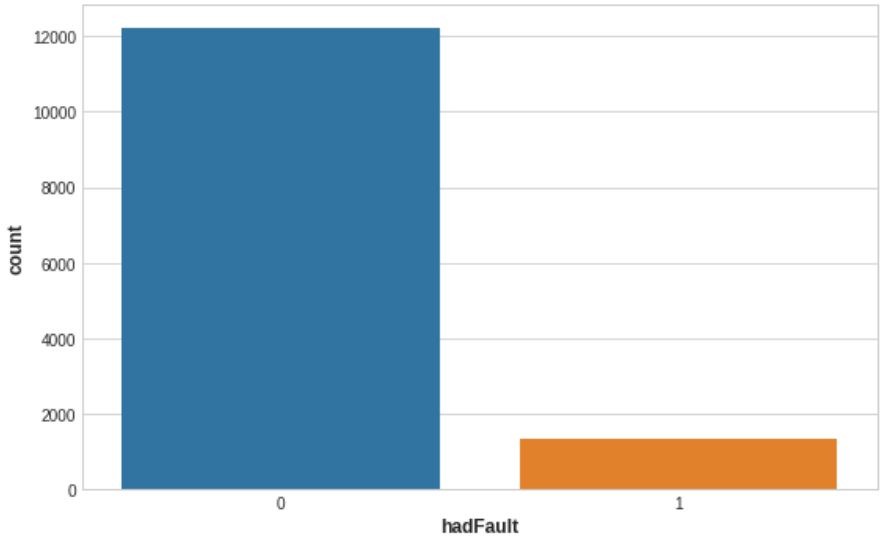

Figure 2.5. The distribution of classes considering just the closest system configuration

The distribution is heavily unbalanced and this fact poses significant, well-known difficulties to the classification task that will be required in the following. On the other hand, unbalanced class distributions are typical and expected when dealing with problems concerning the detection of anomalies, which are indeed defined as infrequent events.

2.3.2 Labelling logics: fault lifetime

The first labelling logic disregarded, for simplicity, a characteristic of the fault signal already mentioned in 1.1.1: a fault signal may well have a short duration or even be

instantaneous, but it could also remain active for a longer period of time, also longer than the time that separates two consecutive *can* packages. In that case, we may consider all the *can* packages that were emitted during the fault lifetime as affected by the fault and label them with 1. On the contrary, we just considered the closest *can* package and ignored the duration of the faults.

In a more advanced phase of our project, a deeper study of the data available in the Data Warehouse made possible to implement this other logic, which is depicted in the diagram in figure 2.6: the *can* packages pointed with the arrows are emitted when a fault signal is active, thus will be labelled with 1. When processing the data needed for this labelling, we also filtered out the faults with *Low* severity, in order to consider the serious faults only. In fact, the severity fault classification has, in principle, the following meaning:

- *Low*: the vehicle still operates and no intervention is needed
- *Medium*: the vehicle still operates, but should be checked as soon as possible from an authorized service point
- *High*: the vehicle is not able to operate and must remain still until assistance is provided

Thus, the filtering of low severity faults is a reasonable preprocessing step, considering the type of faults this category encloses.

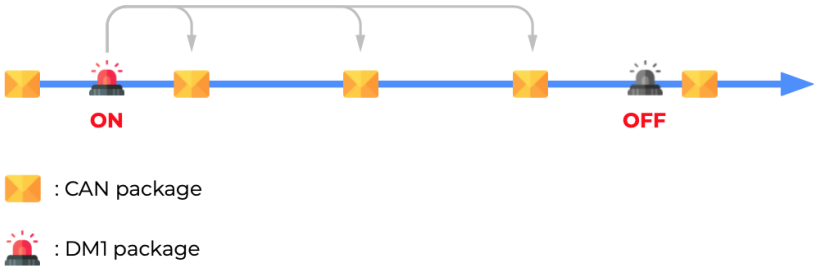

Figure 2.6. The distribution of classes considering fault lifetime

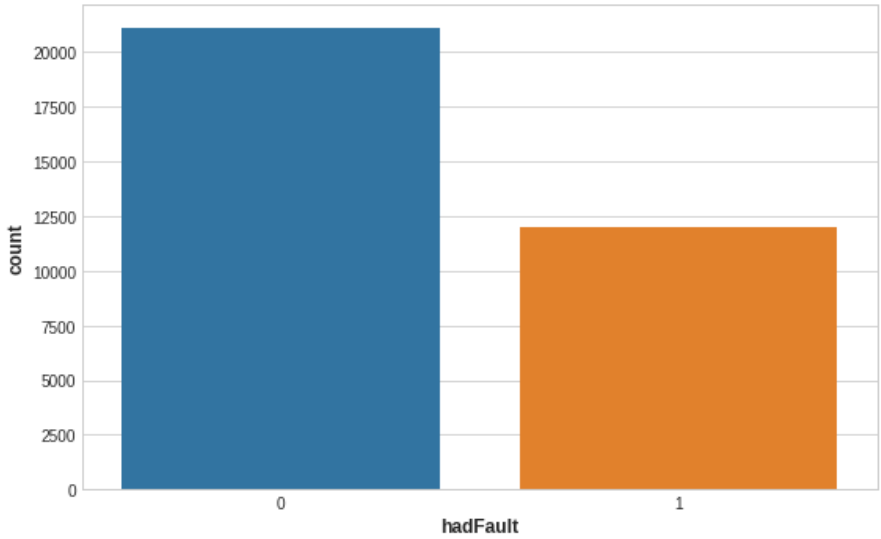

Figure 2.7. The distribution of classes considering fault lifetime

The distribution of classes produced by this logic is shown in figure 2.7.

As the image shows, this labelling logic identifies much more data points as affected by a fault than the previous one, yielding a more balanced classes distribution.
Given the greater ease with which this second logic identifies potentially problematic data points, we decided to exploit it to try and push our learning attempts a little further. Indeed, we may be interested in obtaining rules that characterise not just any fault, but a specific one or a narrower group of issues. Given its greater difficulty to label a sample as a fault, the first logic might complicate the task. On the contrary, the second one seems to be much more convenient. Thus, we employed the second labelling logic to generate the same dataset, but with the labels (0: normal sample, 1: faulted sample) referring to faults related to the specific issue that affect the fleet of vehicles considered (a problem related to the crankcase, briefly mentioned in Chapter 2).

The generated classes distribution is shown in figure 2.8 and it is indeed much more unbalanced than the one obtained in the first case. Nonetheless, a Random Forest fitted on this data achieves slightly better performances than a Random Forest of equal

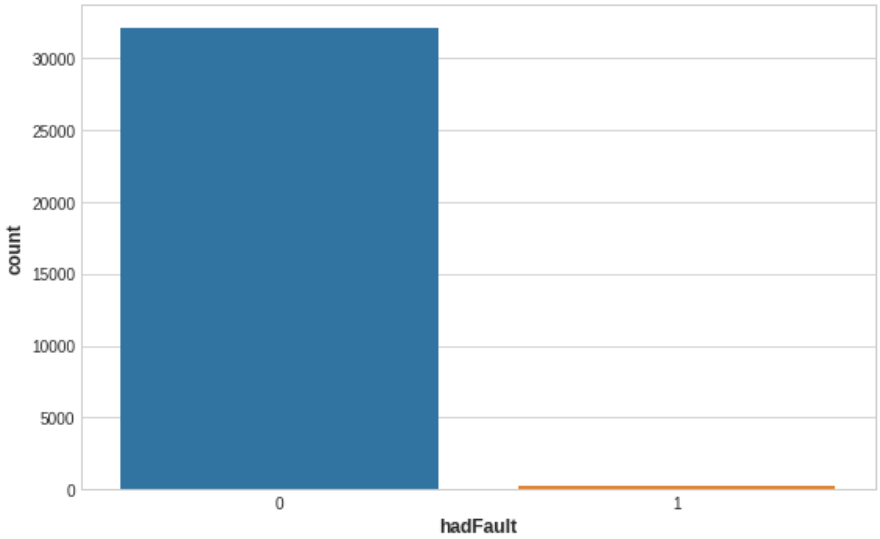

Figure 2.8. The distribution of classes considering fault lifetime (only the ones related to the crankcase issue)

dimensions fitted on the data generated by the first logic, so at least the task difficulty should be approximately the same. This is probably due to the fact that, although there are far fewer of them, the new labels identify a much more clearly defined class of fault. Consequently, the classification algorithm still manages to learn factors that distinguish the aforementioned class.

2.3.3 Final remarks

Labelling logics comparison

The two labelling logics generate very different results and, apparently, the second might seem more effective or, at least, more justified from a conceptual point of view. However, from a practical point of view, the two are not mutually exclusive, but are more or less suitable depending on the type of fault considered. In fact, if we consider a fault (or a category of faults) that usually remains active for prolonged periods of time, the second labelling logic is certainly more suitable. Instead, if the fault is intermittent, that is, it

tends to activate repeatedly for short periods of time, the latter, since it considers the life time of the fault, would have difficulty in correctly labelling the data points. Instead, the first labelling logic would work more effectively. In conclusion, the use of one or the other option should be evaluated according to the specific fault to be characterised.

Fault signal as anomaly detector

Another important point, which we have already mentioned in 1.1.2, is that the *dm1* signal is not a totally reliable indicator of a real issue. The activation of one or multiple error signals does not necessarily mean that the vehicle has an actual problem, but may also be due to a temporary, unproblematic deviation from normal operating conditions or to ambient conditions or to defects in some sensors, *etc.* It may also happen that the actual problem is not the reported one, but a cascade effect brought to the activation of a related fault signal. For this reason, the labels produced by this phase will only serve as a flag to indicate which data points *may* describe an anomalous condition of the system, in order to address a learning algorithm to characterize the features of these statuses. Thus, we used this signal as an indicator of *anomalousness*, so the labelling process discussed may be regarded as an *anomaly detection* phase, specifically tailored to the problem and the data at hand. We will discuss again this matter, when we will locate our work in a more general theoretical framework.

2.4 Visualizations

Exploring and visualizing data is an important task to accomplish before any other kind of processing. In fact, an in-depth knowledge of available data can guide the choice of models and procedures to apply. As our dataset has many features, we will present only a few of the possible graphical views, in order to give an idea at least of its main characteristics.

2.4.1 Variation analysis

Categorical features

Dataset's categorical features all tend to have one most frequent value, while the others are detected more rarely. Figure 2.9 shows two examples.

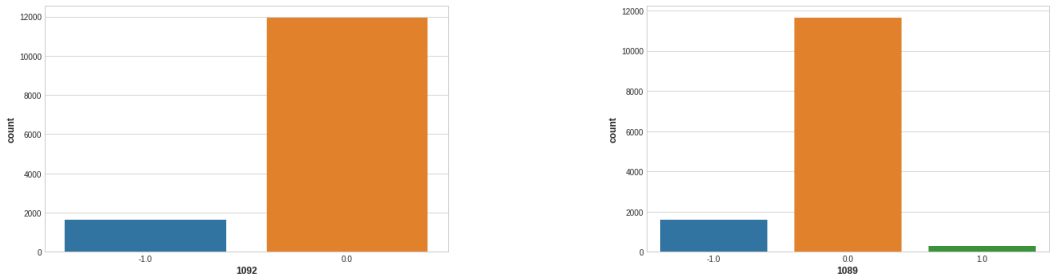

Figure 2.9. Examples of categorical features

Numerical features

Dataset's numerical features have a more diverse range of cases: some features exhibit normal-ish shaped distributions, as shown in figure 2.10, while others have more complex distributions, *e.g.* more skewed or bimodal, as shown in figure 2.11.

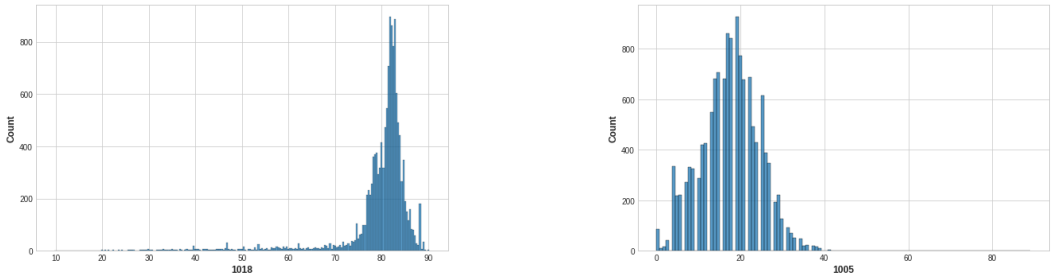

Figure 2.10. Examples of normal-shaped numerical features distributions

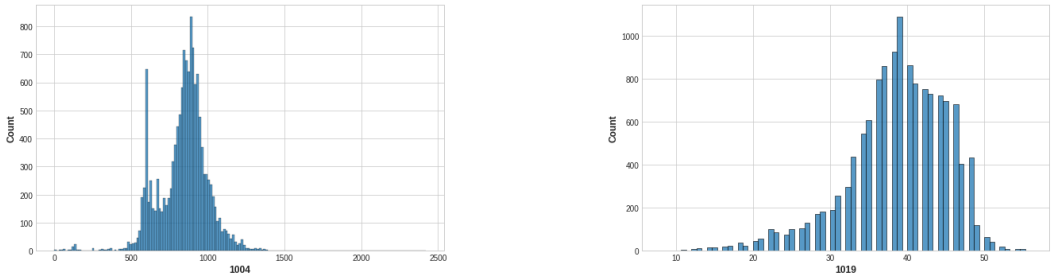

Figure 2.11. Examples of more complex numerical features distributions

2.4.2 Covariation analysis

Within classes distributions

It is useful to display the distribution of a variable broken down by a categorical one. In particular, it would be interesting to discover that the classes of the target variable are related to different distributions of some system parameter. For our dataset, it is sometimes the case, while other parameters do not exhibit different distribution within classes. Figure 2.12 shows a couple of cases of this behaviour for continuous variables with first logic labels, while figure 2.13 considers the second one.

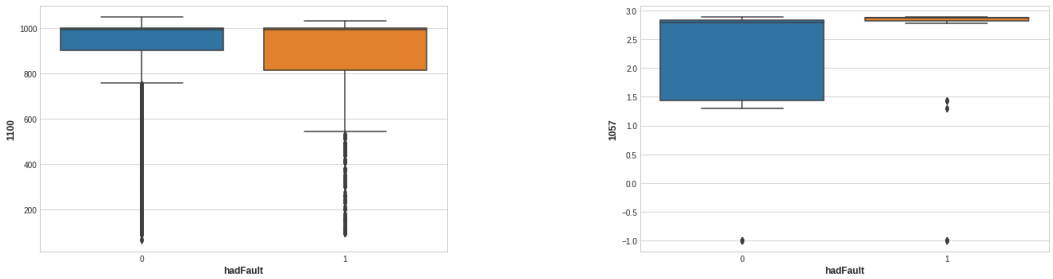

Figure 2.12. Two continuous system parameter's distributions broken down by the target variable values (first labelling logic)

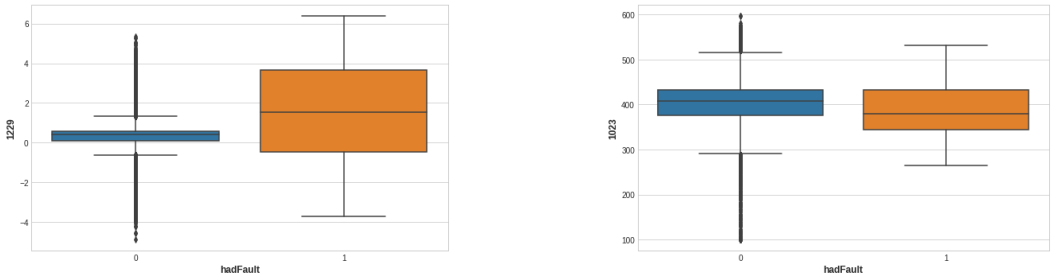

Figure 2.13. Two continuous system parameter's distributions broken down by the target variable values (second labelling logic). The system parameters shown are directly related with the fault considered (*e.g.* parameter 1229 is the pressure in the crankcase)

Categorical variables may also be analysed in this way, showing the frequencies of the possible values broken down by the target variable. Again, some differences can easily be found between normal and anomalous samples groups, as shown in 2.14 and 2.15, respectively first and second labelling logic. At least, this plots all suggest that our labelling approaches manage to divide at least partially different groups of samples.

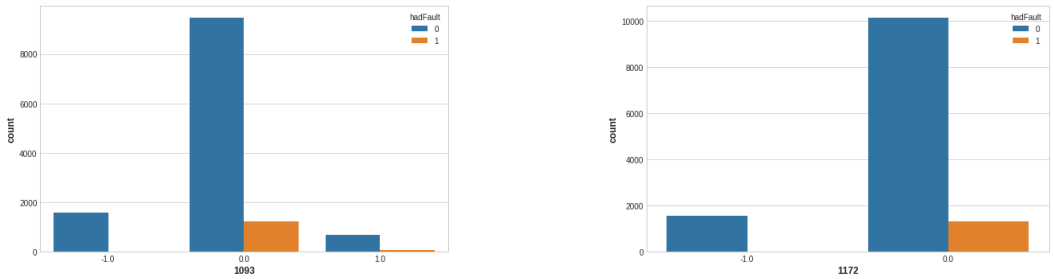

Figure 2.14. Two categorical system parameter's distributions broken down by the target variable values (first labelling logic)

Figure 2.15. Two categorical system parameter's distributions broken down by the target variable values (second labelling logic)

Correlation analysis

In figure 2.16 a heatmap representing the Pearson correlations between the various (numerical) parameters is shown. As can be seen, we have a rather diverse range of cases: some features have very strong (negative or positive) correlations, while many others do not. Figures 2.17 and 2.18 show two of such situations.

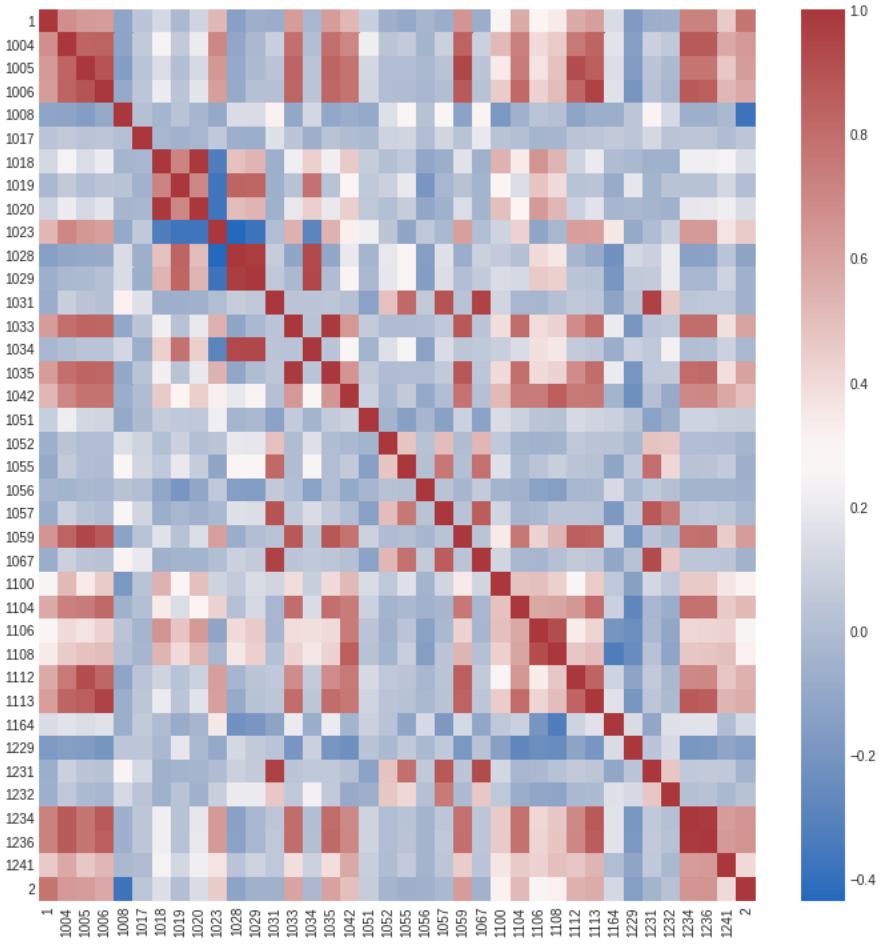

Figure 2.16. Heatmap of correlations between features

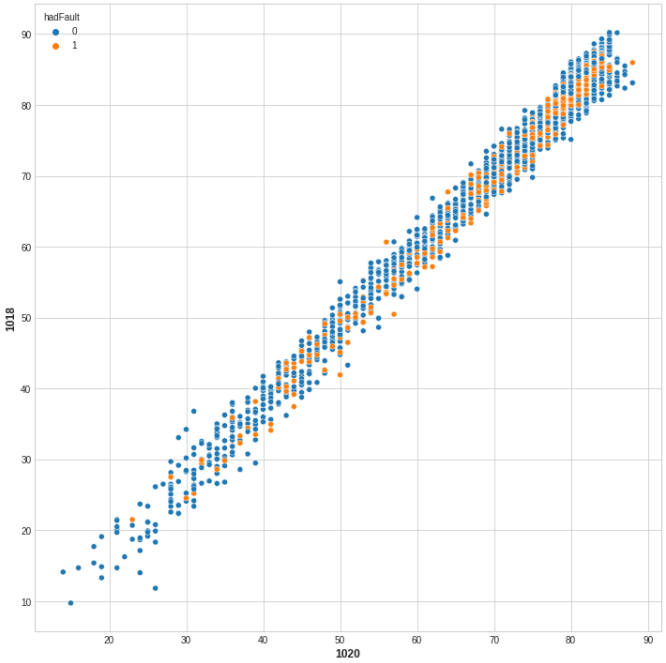

Figure 2.17. Strong positive correlation case

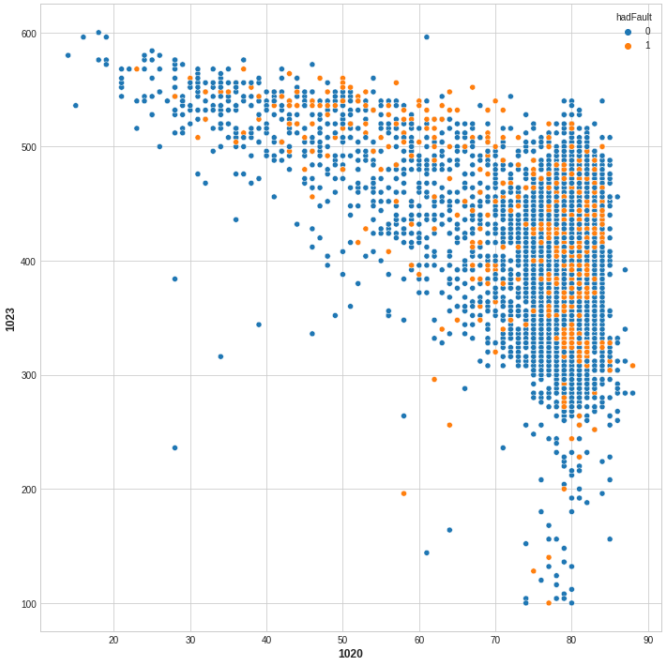

Figure 2.18. Strong negative correlation case

Chapter 3

Anomaly detection foundations

Anomaly detection (or outlier detection) is defined as «the identification of rare items, events or observations which raise suspicions by differing significantly from the majority of the data» [9]. It is important to note that an outlier may not necessarily represent a negative phenomenon (*e.g.* a bank fraud), but could also identify an interesting subset of data (*e.g.* a segment of clients) to be investigated.

Anomaly detection (AD) is applied in a variety of domains, of which fault detection is a very popular one. The effective and precocious identification of issues in an industrial process is critical in order to operate immediate interventions, avoid damages and thus reduce costs. However, an issue that is often overlooked in the context of anomaly detection is that of explaining anomalies, thus sacrificing interpretability. The core part of our project, which will be presented in Chapter 4, may be regarded as an attempt to remedy this lack, adding an explanation step to the identification of an anomaly. For this reason, this chapter introduces the foundations of ML-based anomaly detection. The following discussion is a distillate of [6], which is a more in-depth reference for the interested reader. Section 3.1 introduces some basics concepts, while Sections 3.2, 3.3 and 3.4 present the

main approaches, based respectively on distances, clustering and data-modelling.

3.1 Anomalies and metrics

An anomaly is a substantial variation from the norm, *i.e.* an event or a data point which is particularly different from the majority of usual events or data points. Despite the intuitiveness of this definition, how to move from it to an operational specification of the concept is a non-trivial problem. For example, how the norm should be characterised? And how far from the norm a particular data point should be to be considered anomalous? As there is not a unique definition of similarity between two data points, also the answer to this question is not unique. As a starting point, when applying an anomaly detection algorithm, three possible outcomes must be taken into account:

- *Correct detection*: the algorithm correctly identifies an anomalous event
- *False positive*: the algorithm identifies a normal event as an anomalous one
- *False negative*: the algorithm identifies an anomalous event as a normal one

The terms "positive" and "negative" reflect the fact that, considering anomalous and normal events as class values for a classification problem, the anomaly class is usually referred to as the *positive class*, while the normal class is referred to as the *negative class*. Totally accurate detection systems are typically not possible in real applications, so the objective is usually to minimize false positives and negatives, eventually managing a trade-off between the two quantities. Two important and popular metrics for performance evaluations, based on these concepts, are:

- **Precision**: percentage of correct predictions of the positive class, *i.e.* the number of times the model correctly predicted an anomaly divided by the number of total

anomaly predictions

$$precision = \frac{tp}{tp + fp} \tag{3.1}$$

- **Recall**: percentage of true positive cases detected by the model, *i.e.* the number of times the model correctly predicted an anomaly divided by the number of times the model *should* have predicted an anomaly

$$recall = \frac{tp}{tp + fn} \tag{3.2}$$

Another, less known, performance measure is related to systems that do not classify a data point as normal or anomalous in a binary way, but instead output a ranking of the submitted data points, from the most suspicious to the less likely to be anomalous. Let us suppose that such an algorithm identifies m potential anomalies, while $m_t < m$ is the number of true anomalies. We also let R_i denote the rank of the i-th true outlier in this list. An algorithm should be considered effective if the real outliers occupy the top positions, while the normal instances are at the bottom of the ranking. This idea is expressed by the *RankPower* metric:

$$RP = \frac{m_t(m_t + 1)}{2\sum_{i=1}^{m_t} R_i} \tag{3.3}$$

whose maximum value is attained if all m_t outliers occupy the top m_t positions in the ranking.

As for the definition and characterization of "normality", it should be adapted on a case-by-case basis to the available data:

- For simple (*e.g.* normal) distributions, standard statistics such as mean, median and mode are adequate to characterize the norm, while the distance from it may be regarded as a degree of anomalousness of a data point

- If the involved distributions are more complex (*e.g.* multimodal), mean and similar statistics do not allow the previous simple interpretation. Rather than using a single point, thus, normality may be described by a set of points, for example a set of cluster centroids. The distance of a data point from each cluster would be a possible sign of an anomaly.

- Local characteristics of a dataset may also be used to assess normal and anomalous points. In fact, if a dataset is characterized by areas (in the data space) of high density, when a new data point is located in a low-density location it could be the object of a further analysis.

Multidimensional data pose some additional, well-known complication and require increased care, particularly in the choice of distance measures. Also, a normalization strategy could be required, to eliminate effects due to different magnitudes.
A possible taxonomy of anomaly detection algorithms divides the various approaches in:

1. Distance-based
2. Clustering-based
3. Model-based

The next sections will illustrate and discuss these categories, along with their advantages and drawbacks.

3.2 Distance-based approaches

Given that identifying anomalies amounts at understanding how much a given point is different from the other points, a natural way to address this question is by using the distance in between. This requires to define a measure in the data space and also to decide whether to compare a point to another point or to a group of points. This specifications individuate the different distance-based anomaly detection algorithms.

In this and the following section, we will denote the n-dimensional dataset with $\mathcal{D}$. The characters p, q will denote data points in $\mathcal{D}$, while P denotes a set of points, *i.e.* $P \subset \mathcal{D}$. The distance between two points $p, q \in \mathcal{D}$ is denoted by $d(p, q)$.
The questions to be addressed are essentially three:

- **Measure of anomalousness**: we need a function α such that, given $p \in \mathcal{D}$, $\alpha(p) \in \mathbb{R}$ measures how much anomalous is the point
- **Absolute anomalousness**: this requires to define a threshold θ, related to the function α, such that, if $\alpha(p) > \theta$, then p is anomalous
- **Relative anomalousness**: the function α has to be such that, if p is more anomalous than q, then $\alpha(p) > \alpha(q)$.

In the following, we will first discuss some possible measures of similarity and then address the different distance-based approaches.

3.2.1 Similarity measures

Similarity and distance are two complementary concepts, in the sense that two points in some space may be considered similar if they are close with respect to the distance defined in the space. So, when defining a measure of similarity, also a measure of distance is obtained consequently and vice versa. In standard spaces, some distance measures are very popular.
The simplest one is the Euclidean distance

$$d_2(p, q) = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2}. \tag{3.4}$$

Frequently, some normalization is applied along each dimension before calculations with this distance. For example, a popular choice is to normalize each dimension to have

values in the interval [−1,1] or [0,1]. Euclidean distance is, in fact, a particular case of Minkowski distance

$$d_l(p,q) = (\sum_{i=1}^{n} |p_i - q_i|^l)^{\frac{1}{l}} \tag{3.5}$$

obtained with $l = 2$. A special case of Minkowski distance, defined for $l = \infty$, is

$$d_\infty(p,q) = \max |p_i - q_i| \tag{3.6}$$

that is just the maximum difference in absolute value between the points components. When the dimensions in $\mathcal{D}$ have very different magnitudes and different mutual correlations, an advisable alternative is Mahalanobis distance

$$d_M(p,q) = \sqrt{(p-q)^T S^{-1}(p-q)} \tag{3.7}$$

where S is the covariance matrix between the dimensions of $\mathcal{D}$. When S is diagonal, 3.7 simplifies to

$$d_M(p,q) = \sqrt{\sum_{i=1}^{n} \frac{(p_i - q_i)^2}{\sigma_i^2}} \tag{3.8}$$

where σ_i is the standard deviation along dimension i. Note that Euclidean distance is also a particular case of Mahalanobis distance, obtained when S is the identity matrix. If the dataset dimensions have binary values, *i.e.* $p_i \in \{0,1\}$, Jaccard similarity allows to measure how similar two such points are

$$J(p,q) = \frac{m_{11}}{m_{10} + m_{01} + m_{11}} \tag{3.9}$$

where

m_{11} = number of places where p and q are both 0

m_{10} = number of places where p_i's are 1 and q_i's are 0

m_{01} = number of places where p_i's are 0 and q_i's are 1.

Otherwise, cosine similarity is another possible proximity measure for binary arrays

$$d_c(p,q) = \frac{\sum_{i=1}^{n} p_i q_i}{\sqrt{\sum_{i=1}^{n} p_i^2 \sum_{i=1}^{n} q_i^2}}. \tag{3.10}$$

Finally, Jacob similarity is defined for two sets A and B of data points

$$J(A,B) = \frac{|A \cap B|}{|A| + |B| - |A \cap B|} \tag{3.11}$$

For non-standard cases, tailored measures should be defined, carefully considering which properties this custom distance measure should have.

3.2.2 Main approaches

Once a distance measure has been defined, the next step is to decide against which objects this distance is to be measured. The next sections discuss some possible choices.

Distance to all points

The simplest choice is to measure the distances of each point $p \in \mathcal{D}$ from all the other points and use their sum as an anomalousness metric

$$\alpha(p) = \sum_{q \in \mathcal{D}, q \neq p} d(p,q) \tag{3.12}$$

However, this approach is computationally rather heavy. In addition, it tends to label as anomalous the extremal points in the dataset (*e.g.*, the outermost points on the left and on the right in a group of points on a line).

Distance to nearest neighbour

The opposite choice with respect to the previous one is to look for the point q closest to p and measure the distance in between

$$\alpha(p) = \min_{q \in \mathcal{D}, q \neq p} d(p, q) \tag{3.13}$$

and the most anomalous point is the one farthest from its closest neighbour.

Average distance to k nearest neighbours

Often, measuring the distance from p to all other points in $\mathcal{D}$ is too computationally expensive, while considering just its closest neighbour is not informative enough. Thus, an intermediate choice may be to consider the k nearest neighbours of p and compute the average distance. If we denote with $n_{p,j}$ the j-th nearest neighbour of p, then this measure is defined as

$$\alpha(p) = \frac{\sum_{j=1}^{k} d(p, n_{p,j})}{k} \tag{3.14}$$

A drawback to this approach is that the k parameter must be tuned and greatly influences the results obtained. Usually, some performance measure is defined, related to the task considered, and some visual heuristic is applied (*e.g.* the "elbow" method).

Median distance to k nearest neighbours

An alternative more robust to the inclusion of other points (in particular, points with extreme locations with respect to the others) considers the median distance, instead of

the average. As is often the case in mathematics, this increased robustness comes at the cost of an increased amount of computation, which should be taken into account.

3.3 Clustering-based approaches

Among the most popular anomaly detection approaches, a highly represented category is certainly that of algorithms based on clusters. Also in this case, various choices are possible relating how to identify clusters and how to exploit them to detect anomalies. Clustering may be based on distances or similarities, producing again approaches that are somewhat complementary, often yielding the same results. Nonetheless, the conceptual difference is not to be ignored.
In general, each point $p_i \in \mathcal{D}$ is assigned a degree of membership $\mu(p_i, C_j) \in [0,1]$ to each cluster C_j. There are a few possible ways to assign this numerical property:

- Partitioning algorithms assign each point p_i to one and only one cluster, so that $\mu(p_i, C_j) \in \{0,1\}$ and $\sum_j \mu(p_i, C_j) = 1$.
- Non-partitioning algorithms do not necessarily assign a point to a cluster (and, usually, points which do not belong to any cluster are to be considered anomalous), so $\mu(p_i, C_j) \in \{0,1\}$ and $\sum_j \mu(p_i, C_j) \leq 1$.
- Some algorithms allow points to belong to multiple clusters, yielding $\mu(p_i, C_j) \in \{0,1\}$ and $\sum_j \mu(p_i, C_j) \leq n_C$, where n_C is the number of clusters.
- Finally, fuzzy clustering algorithms assign non-binary degrees of membership $\mu(p_i, C_j) \in [0,1]$, which may be naturally interpreted as probabilities of membership. Usually, the restriction $\sum_j \mu(p_i, C_j) = 1$ is imposed.

3.3.1 Cluster building techniques

In this section, we describe some possible techniques to build clusters.

Nearest neighbour clustering

Algorithms based on nearest neighbours are ubiquitous in Machine Learning and may also be applied to clustering tasks. They are generally based on the idea of local similarity, *i.e.* a point in a space is usually similar to other points nearby. A possible clustering heuristic based on nearest neighbours could be as follows:

i. FOR all points p in a pre-specified set:

   1. Pick a new point p and label it with a new cluster-ID
   2. FOR each point q which is not labelled and is at a distance from a labelled point that is less than a pre-specified threshold:
      a. Label q with its nearest neighbour's label

ii. Label each still unlabelled point with the majority label among its k nearest neighbours

The step i. in the procedure is aimed at creating clusters starting from a set of points distant from each other. After that, step .ii relaxes the local focus, widening the scope to a greater neighbourhood, and labels all points that were not identified by the previous step.

k-means Clustering

Many clustering algorithms are based on the idea of *centroid*, *i.e.* a point (which does not necessarily belongs to the dataset) which is representative of a cluster. Among these algorithms, k-means algorithm is probably the most popular one: it requires k initial centroids, from which the initial group of clusters is built. Then, centroids are iteratively recalculated and points attribution to cluster is updated, until some convergence criterion is satisfied. In particular, the algorithm is as follows:

i. Initialize the cluster centroids $C = \{c_1, \ldots, c_k\}$ with random points in $\mathcal{D}$

ii. WHILE the number of reassigned points is greater than a minimum threshold θ:

1. Assign each point p to the closest threshold, *i.e.* to the cluster C_j, where $j = \arg\min_{j=1,\dots,k} d(p, C_j)$
2. Update each cluster centroid with the mean of the points in the cluster

One can prove that the algorithm minimizes the quantity

$$\sum_{p_i} \sum_{c_j} ||p_i - c_j||^2 \tag{3.15}$$

which is the total sum of the distances between points and the corresponding cluster's centroids.

Despite being very popular, k-means algorithm has two main drawbacks, which make it unsuitable for many problems:

- The clusters resulting from this procedure are hyper-spheres in the data space. In other words, the algorithm assumes that clusters all have this shape, which may not be the case
- The algorithm is greatly dependent on the initial choice of centroids

In addition, the selection of the number of clusters k is a critical step. Usually, it is carried out with heuristics which increase k as long as the incremental improvement outweighs the increased complexity (coming from the higher number of clusters).

Fuzzy clustering

As anticipated above, fuzzy clustering algorithms assign to each point p_i a degree of membership $\mu_{i,j}$ to cluster with centroid $c_j, j = 1, \dots, k$ which belongs to the whole interval [0,1] and is a decreasing function of $d(p_i, c_j)$, *e.g.* $\frac{1}{\exp^{d(p_i,c_j)^2}}$. The underlying idea is that the whole minimization procedure should be carried out assuming that points

closer to the a cluster's centroid should be given more weight than points farther from it. In this view, the quantity to minimize should be

$$\sum_{p_i} \sum_{c_j} \mu_{i,j} ||p_i - c_j||^2. \tag{3.16}$$

The algorithm is a simple modification of the previous one, as follows:

i. Initialize the cluster centroids $C = \{c_1, \ldots, c_k\}$ with random points in $\mathcal{D}$

ii. WHILE the total amount of cluster centroids update is greater than a minimum threshold θ

    1. FOR each p_i compute the degree of membership $\mu_{i,j}$ to each cluster (with centroid c_j) as a decreasing function of $d(p_i, c_j)$.
    2. Normalize the obtained μ's so that $\sum_j \mu_{i,j}$ for each point p_i.
    3. Update each cluster's weighted centroid as

$$c_j = \frac{\sum_i \mu_{i,j} p_i}{\sum_i \mu_{i,j}} \tag{3.17}$$

Agglomerative clustering

Agglomerative clustering starts from considering each point $p_i in \mathcal{D}$ as a (degenerate) cluster, with the point itself as the cluster centroid. At each step, the two clusters with the closest centroids are merged together and the procedure may be carried out until a certain number of clusters or a certain minimum cluster size for each cluster. The algorithm pseudocode is as follows:

i. Initialize the cluster centroids with the dataset $\mathcal{D}$ points.

ii. WHILE the selected convergence criterion is not satisfied yet:

    1. Select the couple of closest centroids

2. Merge the corresponding clusters
3. Update the clusters list and compute the new centroid's distance to all other centroids

Differently from the previous ones, this algorithm is deterministic and does not require the pre-specification of number of clusters k. However, it is more computationally demanding and some decision is still required to select the adequate depth in the resulting hierarchical tree. There are some variants to the basic algorithm, *e.g.* the merging step may be designed to combine more than two clusters at a time.

Divisive clustering

An approach which is opposite to agglomerative clustering consists in starting from a single cluster represented by the whole dataset $\mathcal{D}$ and, at each step, iteratively splitting a cluster from the current clusters list (*e.g.* through k-means with $k = 2$). Given that the computation of centroid distances for each couple of cluster centroids is avoided, this algorithm is sometimes faster than agglomerative clustering.

DBSCAN

Rather than considering distances, a clustering algorithm may be based on local densities. DBSCAN is an agglomerative clustering algorithm based on density defined as number of points in unit volume. The algorithm identifies *core* points, which are points which enclose at least a minimum number $MinPts$ of other points within a neighbourhood of radius r. After a cluster is formed around each core point, if two core points lie at a distance less than r, the corresponding clusters are merged. Points which belong to a cluster and are not core points are called *border* points, while points which do not belong to any cluster are *noise* points and are considered as candidates to be outliers. The advantage of DBSCAN is its capability to build clusters of any shape, not just spherical:

i. FOR each point $p \in \mathcal{D}$:

1. Find all neighbours q within a distance $d(p, q) < r$
2. If p has more than $MinPts$ neighbours, label it as core point

ii. FOR each point $p \in \mathcal{D}$:

1. Assign p to some cluster such that the corresponding centroid c is a distance $d(p, c) < r$, if any.

iii. WHILE the current centroids set contains a couple c_1, c_2 such that $d(c_1, c_2) < r$:

1. Merge the clusters corresponding to c_1 and c_2
2. Update clusters and cluster centroids sets

Being based on a simple definition of density, the algorithm can not be used (at least, not in its basic form) when categorical features are present. Also, it has greater difficulties if the data present widely varying densities. Various modifications have been proposed in literature to address this issues.

3.3.2 Anomaly detection with clusters

Once that clusters are defined, several approaches may be applied to employ these cluster to detect anomalies. We describe some of them in the following parts.

Cluster membership

When employing algorithm which allow points to lie outside of clusters, the obvious way to identify anomalies is to look for points which do not belong to any cluster.

Proximity to other points

For partitioning algorithms, the above criterion is not applicable because each point is assigned to some cluster. Nonetheless, some points may lie far from all other cluster members and this may be a signal of anomaly. To avoid having to compute the distance

of each point from each other point in the cluster, the exploitation of a proxy is advisable. A natural one is the distance from the cluster centroid

$$\alpha(p) = \min_j d(p, c_j) \tag{3.18}$$

where c_j is the centroid of cluster C_j. The distribution of these distances may be analysed, looking for extremal values which may correspond to anomalies in the dataset.

Distance from nearest neighbour

The distance from the cluster's centroid, as noted before, assumes that clusters are symmetrical and may thus fail to find anomalies when the assumption is not verified, at least approximately. In those cases, a more appropriate measure of anomalousness could be the distance of each point p from its nearest neighbour

$$\alpha(p) = \min_q d(p, q) \tag{3.19}$$

This choice presents, again, the inconvenience of having to compute all possible distances. Also, some kinds of anomalousness are not detected. For example, if two points are far from the other points in the cluster, but close to each others, they will not be identified as potential anomalies.

Distance from boundary

Another possible choice, which for example avoids the pitfall highlighted for the previous one, is to consider the distance from the boundary of the closest cluster as a measure of anomalousness. Consider a generic non-partitioning algorithm: for points not belonging to any cluster, we may define

$$\alpha_{out}(p) = \min_j d(p, C_j) \tag{3.20}$$

where $d(p, C_j)$ is the distance from point p to cluster C_j, which is usually defined in the following standard form

$$d(p, C_j) = \min_{q \in C_j} d(p, q). \tag{3.21}$$

On the other hand, we may also want to assess the relative anomalousness of point which lie inside a cluster. In this case, points that are located in the innermost part of the cluster should be considered less anomalous than points closer to the cluster's boundary. We may then measure the anomalousness of cluster points with their distance from the boundary of the cluster itself:

$$\alpha_{in}(p) = \max_{q \in C(p)} d(p, q) \tag{3.22}$$

where $C(p)$ is the cluster the point p is assigned to.

Distances from multiple points

All the approaches discussed above may be modified to consider more points to compute distances from and consequently consider average distance or median distance. As an example, we may compute the distances from p to its k nearest neighbours $N(p) = \{n_1, \ldots, n_k\}$ and assign

$$\alpha(p) = \frac{1}{k} \sum_{q \in N(p)} d(p, q) \tag{3.23}$$

obtaining an anomalousness measure which is more robust to noise. This obviously adds the need of choosing the value for k. A standard tuning frequently applied is $k = 3$.

3.4 Model-based approaches

The last class of anomaly detection approaches we consider consists of techniques based on hypothesizing a mathematical model for the data generation process. In particular, such models represent a concise description of available data and may be built on the basis of relationships between variables, thus obtaining a functional model, or estimating a parametric distribution of data, yielding a distribution model. Data points which do not coincide with this description are considered potential anomalies.

3.4.1 Functional models

The first type of model is expressed by means of relationships between variables. For example, for a dataset $\mathcal{D}$ having two features (*i.e.* two columns in a relational dataset), such a model $M(\mathcal{D})$ could be $x_1+x_2 = c$ or $x_1x_2^d = 0$, where c and d are model parameters. Learning a suitable data model amounts at estimating these parameters on the basis of data by some estimation procedure (*e.g.* least squares fitting). We note that a model defined in this way is deterministic, *i.e.* it does not include any characterization of random fluctuations in data. Once the model has been determined, anomaly detection may be dealt with in the model parameters space or in the data space.

Anomaly detection in the model parameter space

This approach consists in evaluating the influence on the model of a specific data point x by the variation of its parameters. The parameters $\Theta = \{\theta_1, \ldots, \theta_k\}$ of the model of the entire dataset $M(\mathcal{D})$ are compared with the parameters $\Theta' = \{\theta'_1, \ldots, \theta'_k\}$ of the model $M(\mathcal{D}_x)$, where $\mathcal{D}_x \subset \mathcal{D}$ depends on x. A data point with higher influence on the model parameters should be considered more anomalous than a data point with lower influence. A simple measure of anomalousness may take into account the effect on the whole set of

model parameters

$$\alpha(x) = \sum_{i=1}^{k} |\theta_i - \theta'_i|. \tag{3.24}$$

A natural way to define $\mathcal{D}_x$ is $\mathcal{D}_x = \mathcal{D} \backslash \{x\}$. However, if the learning algorithm is robust to outliers, the two models may end up to be the same. Thus, a better choice would be to build $\mathcal{D}_x$ sampling a subset of $\mathcal{D}$ and then explicitly including x.

Anomaly detection in the data space

The data space approach moves the focus from the model parameters to the model predictions to assess the anomalousness of a data point x. As an example, let us consider again the model for a bidimensional dataset $x_1 + x_2 = c$, where c is some estimated parameter. If the point $x = (a, b)$ is such that $a + b = c + 2$, then this point has an error of 2 with respect to the model. Another point with error 0 (*i.e.* the model is perfectly respected) would be considered not anomalous and thus certainly less anomalous than x, while a point with error 4 would be considered more anomalous than x.

Implicit models Implicit models may be applicable when explicit functional relationships are not available. An implicit model is a model of the general form

$$f(x) = 0 \tag{3.25}$$

$$g_i(x) > 0, \quad i = 1, \ldots, k \tag{3.26}$$

The extent to which the various constraints are violated may be employed as a measure of anomalousness. An important question is how to attribute relative weights to these components. In general, the answer to this question depends on the problem: in some cases, expert knowledge may help to assess the relative importance of equality and inequalities; in other cases, this information is not available and the weights have to be

determined by the examination of data.

Explicit models An explicit model states a relationship between dependent variables $y = (y_1, \ldots, y_n)$ and independent variables $x = (x_1, \ldots, x_m)$. The relationship is expressible in closed form as $y = F(\Theta, x)$, where F is the model for $\mathcal{D}$, Θ represents the model parameters and $(x, y) \in \mathcal{D}$ is a data point. A definition of anomalousness can be immediately based on such a model

$$\alpha(x, y) = |y - F(\Theta, x)|. \tag{3.27}$$

Clearly, other distance measures can be employed to measure the deviation of y from the model prediction. As an aside, we highlight the critical importance of evaluating the imprecision or uncertainty of the model, in order to avoid confounding model noise with significant deviations from the model's predictions.

3.4.2 Distribution models

Among distribution models, we can distinguish models based on a parametric distribution estimation and models based on regression.

Parametric distribution estimation

The view of a dataset as a sample from some distribution to be estimated is the preferred one from a statistical point of view and naturally encompasses the presence of some variation in the data itself, being, in this sense, less strict with respect to a functional model. As an example, for a univariate dataset which is believed to follow a normal distribution, the model is

$$f(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{3.28}$$

and the exact distribution is found estimating the parameters μ and σ.

Coming to the evaluation of anomalousness, some caution is required. In fact, as we said above, some variability of the data points from the mean μ is included in the definition of the distribution itself, thus it is not a suitable indicator of an anomaly. A few alternatives may be considered:

- Use the distance from the mean $|x - \mu|$ as a measure of *relative* anomalousness of data points
- Use the properties of the distribution to evaluate the probability of occurrence of a specific data point, as a measure of relative non-anomalousness
- Estimate the distribution parameters with and without a specific point, to evaluate its anomalousness through the variation of them, as we discussed for the parameters space approach for functional models

In some cases, a single distribution may not be adequate to describe the way in which data tend to group together. In those cases, *mixture* distributions could fit better the dataset considered.

Regression models

Regression algorithms are united by the use of gradient based optimization to minimize some error measure, iteratively modifying the model parameters.

Linear and non-linear regression As the name suggests, linear regression hypothesises a linear relationship between a dependent variable y and a vector of independent variables $x = (x_1, \ldots, x_p)$, as

$$f(x) = a_0 + \sum_{j=1}^{p} a_j x_j \tag{3.29}$$

and the a_j parameters are estimated minimizing some measure of error. The standard one is the mean squared error

$$MSE = \sum_i (y_i - f(x_i))^2 \tag{3.30}$$

given its differentiability, which makes it suitable for gradient based optimization. In fact, at each step the parameter a_j is modified by the quantity

$$\Delta a_j = \eta \frac{\partial MSE}{\partial a_j} \tag{3.31}$$

where η is commonly referred to as the *learning rate*. Non differentiable error measures, *e.g.* absolute deviation, are usually avoided. Also, least squares minimization has the advantage of being characterized by a closed form solution, so optimization is not even actually required.
On the contrary, non-linear models do not have this advantage and must be estimated by method such as the non-linear least squares. However, they allow to model a much richer variety of phenomena.

Kernel regression and SVM Kernel regression is a non-parametric technique to estimate the conditional expectation of a random variable Y with respect to a random variable X

$$\mathbb{E}[Y|X] = m(X) \tag{3.32}$$

where $m(\cdot)$ is an unknown function. The latter is modelled by a locally weighted average function, where weights are determined by a *kernel* function. This technique allows to estimate a non-linear relationship between X and Y. The first proposal for

estimating $m(\cdot)$ is the Nadaraya-Watson estimator

$$\hat{m}_h(x) = \frac{\sum_{i=1}^{n} K_h(x - x_i) y_i}{\sum_{i=1}^{n} K_h(x - x_i)} \tag{3.33}$$

where $K_h(\cdot)$ is a kernel function and h is a smoothing parameter called *bandwidth*. The choice of Gaussian kernel functions is typical with Support Vector Machines (SVMs).

Chapter 4

Rule Extraction

The core part of our project was the implementation of a rule extraction algorithm. In this case, a rule is just a set of conditions that identify a problem on a monitored vehicle, based on the system parameters values. To a domain expert, a rule also yields a comprehensible description of the problem, being thus handy in the troubleshooting phase. A data-driven pipeline aimed at automatically building these rules would greatly increase the efficiency of the whole rule definition process, leaving only the validation phase to the professionals of the client company. Moreover, rules constitute the input of the rule engine and may also be a matter of study and analysis, in order to gain insights about issues on the vehicles.
Framing it into the previous chapter, this algorithm could be considered as an extension to the anomaly detection phase, consisting of an anomaly *explanation* phase. In fact, anomaly detection systems are usually not designed to provide for an explanation of the anomalies themselves, making them so-called "black boxes". Our solution tries to move a step further, providing a characterization of the detected anomalies in the form of a rule, as defined above.
In our specific application, the anomaly detection was performed, in some sense, through the DM1 signal in the labelling phase, as described in 2.3. In fact, the DM1 signal, although not entirely accurate and reliable as discussed above, is the result of monitoring

carried out by the vehicle's on-board sensors, the purpose of which is precisely to monitor its operation and report any anomalies. Of course, for other applications, the anomaly detection phase may be conducted with one of the methods presented in Chapter 3 or with some other tailored procedure, the result of which may be cascaded to our explanation algorithm.

In this chapter, we first discuss, in section 4.1, the independent components of the rule extraction pipeline, which are a classification algorithm and an optimization procedure. Then, the main algorithm is described in 4.2 and results are presented and analysed in 4.3. Finally, some possible directions of further improvements are defined in 4.4.

4.1 Prerequisites

The rule extraction algorithm we discuss is based on two main components: a classification model, the Random Forest Classifier, and an optimization algorithm, the random-restart Stochastic Hill Climbing. In fact, it is not a general rule extraction algorithm, but it is designed ad-hoc for tree-based classifiers. Thus, we will first introduce and discuss this topics, so that we will then be able to focus on the rule extraction algorithm itself.

4.1.1 Tree-based methods

The main idea of tree-based methods is to partition the feature space into rectangles and then fit a constant model on each one of them. They are extensively used models in practice, especially in their ensemble version. We will first explain **regression trees** and then discuss the simple extension to **classification trees**. Finally, we will explain the bagging procedure that leads to **Random Forests**. Our discussion of the models and their fitting procedures follows the lines of [2] (from which we borrow a couple of images).

Introductory example

Let's suppose to have a continuous response variable Y and two predictors X_1 and X_2. A regression tree models the regression function as a piecewise-constant function. To this aim, at the first step of the fitting procedure a variable and a split-point are chosen in order to obtain the best fit on training data, leading to two regions of the feature space. At the next step, one of these regions is split again with the same procedure and so on. As an example, consider figure 4.1.

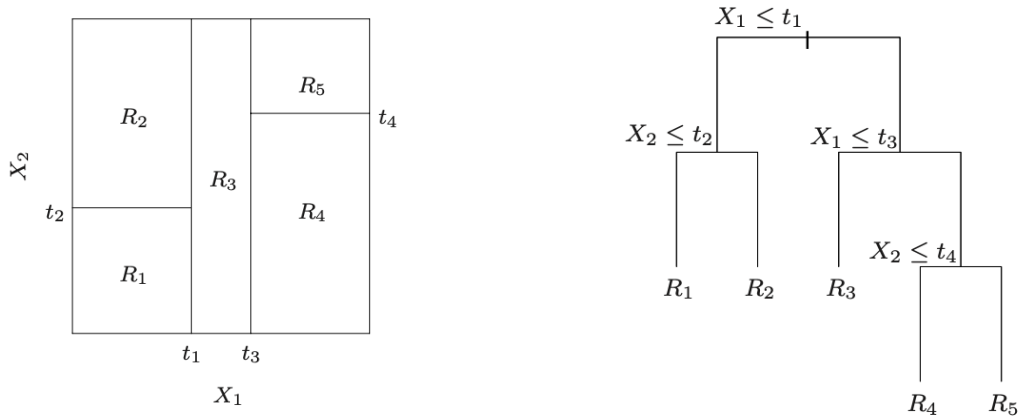

Figure 4.1. Example of regression tree (image from [2])

In the left figure, a partition of the feature space is shown. The first split is made on $X_1 = t_1$ and produces the regions $X_1 \leq t_1$ and $X_1 > t_1$. Then, the first region obtained in the first step is split on $X_2 = t_2$ and so on. This particular way of consecutively splitting the feature space is called "binary splitting" and, while it ensures that the partition set is composed of rectangles, avoiding more complex regions, it also allows to represent the same partition with a tree structure, as the one depicted in the right figure of 4.1. In this simple example, the resulting partition has five regions, denoted with $R_i, i = 1, \ldots, 5$, each one corresponding to a leaf of the related tree. The regression model based on these

regions predicts Y with a constant c_m on each region, as

$$\hat{f}(X) = \sum_{i=1}^{5} c_m I\{(X_1, X_2 \in R_m)\} \tag{4.1}$$

where c_m is the mean response in the region R_m.

Clearly, the representation of the partition in the feature space is possible with two (at most three) inputs only, while the tree representation is always available, making the model easy to interpret.

In the following, we will discuss the building process of the partition.

Regression Trees

Let us suppose that our data consists of N samples (x_i, y_i) for $i = 1, \ldots, N$, where $x_i = (x_{i1}, x_{i2}, \ldots, x_{ip})$. After $M - 1$ splits, the feature space is partitioned in M regions $R_1, \ldots, R_M$ and the current model for the regression function is

$$f(X) = \sum_{i=1}^{M} c_m I\{(x \in R_m)\} \tag{4.2}$$

It can be proved that, if the criterion followed by the algorithm is the minimization of the sum of squares $\sum(y_i - f(x_i))^2$, the best value for c_m is

$$\hat{c}_m = average(y_i | x_i \in R_m) \tag{4.3}$$

as mentioned above in our introduction.

Now, consider a splitting variable j and a split point s, which identify the two semi-planes

$$R_1(j, s) = \{X | X_j \leq s\} \tag{4.4}$$

$$R_2(j, s) = \{X | X_j > s\} \tag{4.5}$$

Thus, j and s must solve

$$\min_{j,s} \quad [\min_{c_1} \sum_{x_i \in R_1(j,s)} (y_i - c_1)^2 + \min_{c_2} \sum_{x_i \in R_2(j,s)} (y_i - c_2)^2] \tag{4.6}$$

The inner minimizations, which just represent the minimization of the squared error in the two regions generated by splitting for the variable j at the split point s, are solved by the value in equation 4.3, so for each variable j the best split point can be found very quickly. Thus, the algorithm has to scan through all the variables, find the best split point for each one and finally select the best (j, s) couple with respect to the total sum of squares. The splitting procedure is then repeated on each of the generated regions.
Clearly, a stopping criterion has to be adopted. One could require the algorithm to stop when the decrease in the total sum of squares determined by the next split does not exceed some predetermined threshold. However, this strategy may preclude to obtain some better error reduction in the following steps. Usually, the algorithm is stopped when the tree reaches a fixed maximum depth or some minimum number of nodes and, optionally, the obtained tree is pruned. The most popular pruning strategy is the *cost-complexity pruning*. Denoting the complete tree with T_0, consider a subtree $T \subset T_0$ obtained through pruning and be $|T|$ the number of terminal nodes in T. Let us define the cost-complexity criterion as

$$C_\alpha(T) = \sum_{m=1}^{|T|} N_m Q_m(T) + \alpha|T| \tag{4.7}$$

where

$$N_m = \#\{x_i \in R_m\} \tag{4.8}$$

$$\hat{c}_m = \frac{1}{N_m} \sum_{x_i \in R_m} y_i \tag{4.9}$$

$$Q_m(T) = \frac{1}{N_m} \sum_{x_i \in R_m} (y_i - \hat{c}_m)^2 \tag{4.10}$$

being them, respectively, the number of training samples, the mean response and the mean squared error in region R_m. Frequently, given that each region R_m is related to a terminal node m, the value $Q_m(T)$ is also referred to as a measure of node impurity. In fact, a smaller value of this quantity in region R_m is obtained if the samples falling in that region have a more similar value for the response Y (with the smallest possible value being obtained when the samples have all the exact same value of the response). Thus, in some sense, these samples are more akin to each other and the node is purer. The objective of the algorithm is actually to segregate the training samples in regions as pure as possible, so it can predict on new samples with greater confidence.

However, it should be clear that the criterion in 4.7 considers both the goodness of fit, through the first term, and the model complexity, represented by the second term. The α parameter modulates the relative importance of the two components. The pruning algorithm finds, for each value of α in some range, the subtree $T_\alpha \subset T_0$ that minimizes 4.7. For large values of α, it will be obtained a smaller tree with higher fitting error and viceversa. It can be shown that, for a given value of α, there is a unique subtree T_α that minimizes $C_\alpha(T)$ and this tree can be found with a procedure called *weakest link pruning*. In broad terms, the procedure iteratively collapses the internal node that results in the smallest increase of the sum of squares, until the root node is obtained. The sequence of trees produced contains T_α.

Classification Trees

If the response variable has discrete values $1,2,\ldots,K$ representing classes, the algorithm needs some pretty straightforward modifications. In fact, we have to select a different measure of node impurity. With similar notation as above, in terminal node m we denote

$$\hat{p}_{mk} = \frac{1}{N_m} \sum_{x_i \in R_m} I(y_i = k) \tag{4.11}$$

the proportion of class k observations in node m. The notation $\hat{p}$ is to suggest that we could interpret the proportion as an estimated conditional probability, as

$$\hat{p}_{mk} = \mathbb{P}[y_i = k | x_i \in R_m] \tag{4.12}$$

An observation in node m is classified with class $k(m) = \arg\max_k \hat{p}_{mk}$, the majority class in the corresponding region. Most popular measures of node impurity for classification trees are presented in figure 4.2.

Misclassification error	$\frac{1}{N_m} \sum_{i:x_i \in R_m} I(y_i \neq k(m)) = 1 - \hat{p}_{mk}$
Gini index	$\sum_{k \neq k'} \hat{p}_{mk}\hat{p}_{mk'} = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk})$
Cross-entropy	$-\sum_{k=1}^{K} \hat{p}_{mk} \log \hat{p}_{mk}$

Figure 4.2. Node impurity measures for classification trees

Actually, misclassification error is not differentiable, so Gini index and cross-entropy are usually preferred and are the most used measures. This is shown in figure 4.3.

Strengths and limitations of trees

As discussed in the previous sections, regression and classification trees have a rather simple fitting procedure and generate interpretable models of the relationship between response and predictors. They have other advantageous peculiarities that we did not

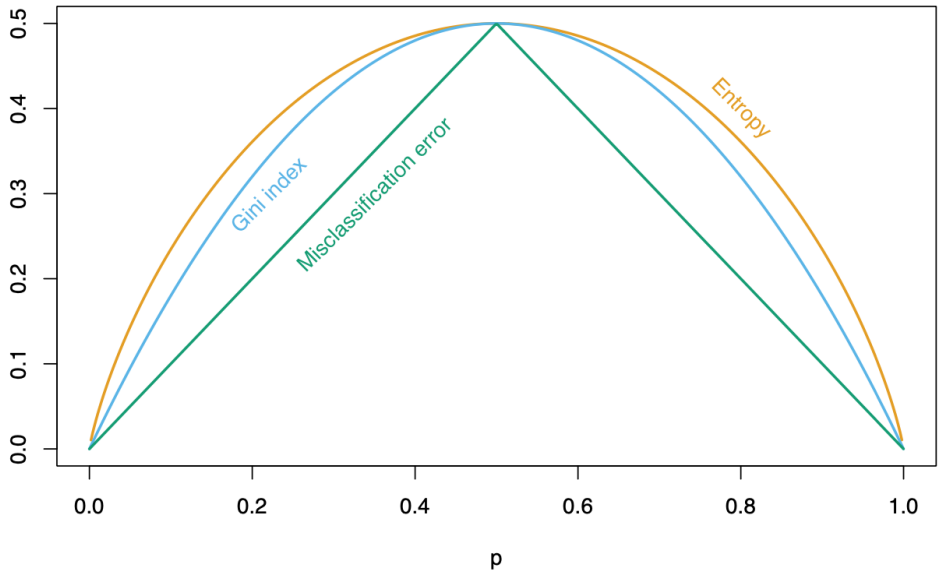

Figure 4.3. Comparison between impurity measures for classification (image from [2])

mentioned (for example, it is possible to mitigate the problem of missing values on some predictor using a surrogate predictor, *e.g.* a predictor that best mimics the split on the first one). Nonetheless, some drawbacks must be considered, probably the major one being a pronounced propensity to overfitting, especially in presence of categorical predictors having a high number of possible categories.

The need to reduce the variance of the fitted model is the reason why ensembles of trees are usually employed, rather than single trees. An ensemble of trees is called a *Random Forest* and we will discuss them in the following section.

Random Forests

Bagging (or bootstrap aggregation) is a general technique used to reduce the variance of an estimated model and it works especially well with decision trees. Random forests is a modification of bagging procedure aimed at building a large set of decorrelated trees.

The basic idea of bagging comes from a basic probabilistic property: the average of B i.i.d. random variables, each with variance σ^2, has variance $\frac{1}{B}\sigma^2$. If these variables are

just identically distributed, with positive correlation ρ, their mean has variance

$$\rho\sigma^2 + \frac{1-\rho}{B}\sigma^2 \tag{4.13}$$

Bagging consists in fitting a sequence of unbiased, identically distributed models on bootstrapped samples of the training data and averaging them (or casting a majority vote in the classification setting) to obtain a variance reduction. However, equation 4.13 shows that the correlation between the individual models limits the potential improvement. The idea of Random Forests is to reduce the correlation of the fitted trees through feature sampling to improve the reduction of variance. Feature sampling just means that, during the building process, at each step the algorithm does not consider every possible feature for splitting, but draws a subsample of m features to be considered. A typical value for m is $\lfloor\sqrt{p}\rfloor$, where p is the number of features.
The complete algorithm is as follows:

1. For $b = 1$ to B:

   (a) Draw a bootstrap sample from the training data

   (b) Build a tree T_b on the sample by applying the following step on each terminal node until some maximum depth or minimum number of nodes is reached:

      .i Select m predictors

      .ii Compute the best variable and split point from the sampled features

      .iii Split the terminal node in two children nodes

2. Return the set of trees $\{T_b\}_{b=1,\dots,B}$

The prediction from the random forest on a new data point x, being $T_b(x)$ the prediction of the b-th tree on x for regression or $C_b(x)$ the class prediction of the b-th tree for classification with classes $K = 1,2,\dots,N$, is

Regression: $$\hat{f}_{rf}^{B}(x) = \frac{1}{B}\sum_{b=1}^{B} T_b(x) \tag{4.14}$$

Classification: $$\hat{C}_{rf}^{B}(x) = \underset{k=1,\ldots,N}{\arg\max} \sum_{b=1}^{B} I(C_b(x) = k) \tag{4.15}$$

Smaller values of m will tend to reduce the correlation between pairs of trees of the forest and thus reduce the estimator variance. Nonetheless, this will also increase bias, so the potential benefit could be offset by this trade-off, which has to be taken into account consequently. An extreme version of the algorithm (called, in fact, *Extremely randomized trees*) increases to randomness injected in the building process by evaluating split values drawn at random. Again, the benefit coming from randomizing is not infinite. In any case, not all estimators can be improved with bagging and trees probably represent the case in which its application proves the most effective. In the next section, we will briefly analyse variance and bias of the Random Forest estimator.

Estimator variance and bias

Let us focus, for simplicity, on the Random Forest regressor, to avoid some complications that arise in the classification case. We recall that the estimator is

$$\hat{f}_{rf}(x) = \frac{1}{B}\sum_{b=1}^{B} T(x; \Theta_b(\mathbf{Z})) \tag{4.16}$$

where x is a data point and $T(x; \Theta_b(\mathbf{Z}))$ is the b-th tree of a random forest composed of B trees. $\mathbf{Z}$ is the bootstrapped sample on which the tree is grown and Θ_b contains the characterization of the tree in terms of split features, split points and terminal values, thus depends on $\mathbf{Z}$. Applying the limit operator, we obtain the limiting form of the estimator

$$\hat{f}_{rf}(x) = \mathbb{E}_{\Theta|\mathbf{Z}} T(x; \Theta(\mathbf{Z})) \tag{4.17}$$

where Θ is conditioned on $\mathbf{Z}$ because we first draw the bootstrapped random sample and then grow the tree on it, drawing features at random on each step. The estimator variance is easily obtained from 4.13

$$\mathrm{Var}\hat{f}_{rf}(x) = \rho(x)\sigma^2(x) \tag{4.18}$$

One may be confused about what the correlation and variance of 4.18 refer to. When computing the expected value and variance of the random forest estimator, we are averaging trees, so

- $\rho(x)$ is the sampling correlation between the predictions that any pair of trees considered in the averaging yield on x:

$$\rho(x) = \mathrm{corr}[T(x;\Theta_1(\mathbf{Z})), T(x;\Theta_2(\mathbf{Z}))] \tag{4.19}$$

  In other terms, $\rho(x)$ is not the correlation between a couple of trees of a given random forest, but is the theoretical correlation between two random forest trees evaluated at x, where the random variation comes from the distribution generated from the tree building process through the sampling steps (either of $\mathbf{Z}$ and the features)

- $\sigma^2(x)$ is the sampling variance of the prediction that any randomly drawn tree yields on x:

$$\sigma^2(x) = \mathrm{Var}[T(x;\Theta(\mathbf{Z}))] \tag{4.20}$$

Regarding the correlation term, as we discussed before, it tends to decrease when decreasing m. In fact, if we reduce the number of randomly drawn features considered at each split, the probability that any two trees use the same splitting variables will diminish itself, leading to less similar predictions on x.
Focusing on the variance of a single tree, it can be decomposed in two components

$$\mathrm{Var}_{\Theta,\mathbf{Z}} T(x;\Theta(\mathbf{Z})) = \mathrm{Var}_{\mathbf{Z}} \mathbb{E}_{\Theta|\mathbf{Z}} T(x;\Theta(\mathbf{Z})) + \mathbb{E}_{\mathbf{Z}} \mathrm{Var}_{\Theta|\mathbf{Z}} T(x;\Theta(\mathbf{Z})) \tag{4.21}$$

$$= \mathrm{Var}_{\mathbf{Z}} \hat{f}_{rf}(x) + \mathbb{E}_{\mathbf{Z}} \mathrm{Var}_{\Theta|\mathbf{Z}} T(x;\Theta(\mathbf{Z})) \tag{4.22}$$

The first term is the sampling variance of the random forest estimator, which decreases when m decreases. The second term is the (expected) variance *within* the random sample $\mathbf{Z}$, so is a result of the randomization of the growing process and increases when m decreases (note that decreasing the number of features the algorithm can consider when splitting increases the variability of the generated trees). It can be shown ([2] does it with a simple simulation), that the variance of an individual tree does not change much over the range of the possible values of m. Thus, equation 4.18 allows to conclude that the variance of the ensemble is lower than the variance of the single tree.
As for the bias, the random forest has the same bias as an individual randomly sampled tree

$$\mathrm{Bias}(x) = \mu(x) - \mathbb{E}_{\mathbf{Z}} \hat{f}_{rf}(x) = \tag{4.23}$$

$$= \mu(x) - \mathbb{E}_{\mathbf{Z}} \mathbb{E}_{\Theta|\mathbf{Z}} T(x;\Theta(\mathbf{Z})) \tag{4.24}$$

Note that this tree will typically have a greater bias than an unpruned tree grown over the same bootstrapped sample $\mathbf{Z}$. In fact, randomized feature selection and the reduction of the sample space pose more restrictions, thus increasing bias. Therefore, the improvement obtained by the random forest comes uniquely from decreasing the variance of the prediction.

4.1.2 Random-restart Stochastic Hill Climbing

A crucial part of the rule extraction algorithm faces the problem of exploring a huge set of objects (≈ hundreds of thousands of them, in the standard case) with the objective of finding a subset that is (sub)optimal with respect to an objective function. We recall that, for a set $\mathcal{S}$ composed of N elements, the power set $\mathcal{P}(\mathcal{S})$ of $\mathcal{S}$ has 2^N elements. The dimensionalities produced by our problem preclude the possibility of exhaustively exploring this set, leaving a suitable heuristic technique as the only feasible possibility. In particular, the algorithm we will use belongs to the class of *Local Search algorithms*, which are characterised by the fact that they search from a start state to neighbouring states, ignoring the path followed and the states visited to reach the current state. Our objective function should be maximized and the process of searching for a global maximum is called *hill climbing*. In the following section, we will first introduce the Hill Climbing algorithm, then present its stochastic version and discuss the idea of *random restarts*.

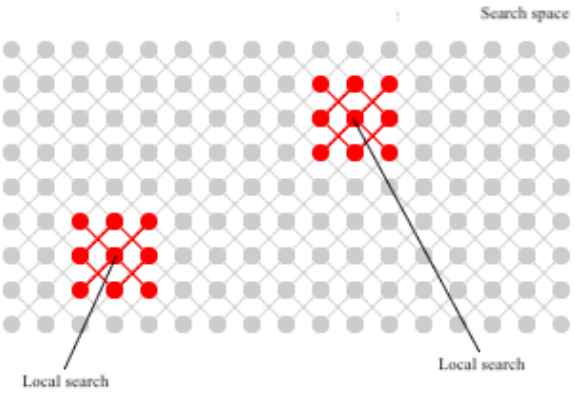

Figure 4.4. Depiction of local search (image from [7])

Hill climbing algorithm

The vanilla version of the algorithm proceeds in a very simple way: it keeps only track of the current state, searches for the immediate neighbour with the highest value (of the objective function) and moves to it. So, the algorithm searches for the direction of local steepest ascent. The search process is stopped when the algorithm cannot find an immediate neighbour with a higher value.
The complete algorithm is as follows:

1. Initialize current state with the initial state
2. While **TRUE**:
   (a) Select the immediate neighbour with the highest value
   (b) **If** the value of the selected neighbour is greater or equal to the value of the current state, update the current state with the neighbour, **else STOP**

Note that the algorithm does not look beyond the immediate neighbourhood of the current state, so, in this sense, is a greedy algorithm. Clearly, it is unlikely that a greedy approach will lead to an optimal solution, but in some cases it could be the only viable approach. Nonetheless, this kind of algorithms often succeed in finding a reasonable solution, though not the best one. This approach obviously has its weaknesses. In particular, the following structures may prematurely end the algorithm in an unsatisfying solution or even stuck it in an infinite search (if we do not impose additional stopping criteria):

- *Local maxima*: a local maximum is a point in the search space that is higher than its neighbours, but it is not a global maximum. When the algorithm gets close to a local maximum, it will typically get attracted and will stop on it. Local maxima may or may not represent a good solution.
- *Ridges*: a ridge is a sequence of local maxima, all next to each other. Usually, local search algorithms have difficulties to navigate this kind of structures and may continuously jump from one to another without being able to stop.

- *Plateaus*: a plateau is a flat area in the search space. In some cases, the plateau could be an area of local maximum, while in other cases it could be contiguous to a direction of further improvement. In either cases, if the flat area is large enough, the algorithm may wander indefinitely.

Also, note that this version of the algorithm assumes that executing a complete *local* search is a feasible task, so that we can inspect each immediate neighbour in order to choose the best one. If each state has many neighbours (*e.g.* thousands of neighbours), also an exhaustive local search may become infeasible. In the next paragraph, we will address this situation.

Stochastic hill climbing and random restarts

As mentioned above, in some spaces a local search of the surroundings may be computationally too expensive or even unfeasible. In this cases, we could benefit from a stochastic variation of the previous algorithm: if we can efficiently distinguish the uphill moves from all the possible moves, we may choose at random from among them. An improvement would always be assured, although the convergence of the algorithm will typically be slower. The probability of selecting a particular move could be proportional to the steepness of the corresponding direction. This variation is called *stochastic hill climbing*.
Making a further step, we may even generate moves at random until we find a state better than the current one. This implementation is called *first-choice hill climbing* and is generally used when each state of the search space as many neighbours.
A last variant is meant to tackle the problem of local maxima: the idea is to start many hill climbing searches, each time from a randomly generated state (thus the name *random restart hill climbing*), in order to compare the solutions obtained and pick the best one. As an aside, note that random restart hill climbing will find the global maximum with probability equal to 1: in fact, it will eventually be initialized in the global maximum itself.

Of course, depending on the problem, the variations we discussed may also be combined. In fact, our implementation will exploit a mix of the methods presented in this section.

4.2 Rule extraction algorithm

The central part of our project concerned the construction of an algorithm for the extraction of rules from a Random Forest, fitted on the data presented in Chapter 2. As discussed in Chapter 1, these rules are at the heart of the operation of the main function of the fault monitoring system. Currently, they are designed manually, based on domain knowledge about the engines. In addition to the inefficiency of this process, already discussed, this means that criticalities that are not known to the client company's professionals can not be addressed by the rule engine. These problems can only be noticed and dealt with once they have occurred a number of times and have caused significant damage. On the other hand, a data-driven system that constantly monitors the data sent by vehicles could detect the problem much earlier, reducing costs and improving service to end customers. Moreover, the rule extraction system provides an explanation to this issues and could be used to address specific faults (or categories of faults) and the generated rules may be further analysed and studied (either with other statistical or ML-based methods or through the domain knowledge about the vehicles) to acquire insights about a particular malfunctioning.

4.2.1 General description

As discussed in 4.1.1, Random Forest is a learning method consisting in fitting multiple *base learners* (or *weak learners*, which in this case are Decision Trees), enforced to be as decorrelated as possible, on bootstrapped samples of data. We recall that each tree in the RF is built from data by a splitting procedure. This splits are just IF-ELSE conditions: if a certain feature's value of a sample is under some learned threshold, the latter follows a branch of the tree; else, it follows the other branch. This process is applied multiple

times, until some criteria, discussed in the previous sections, are satisfied. An example is shown in figure 4.5.

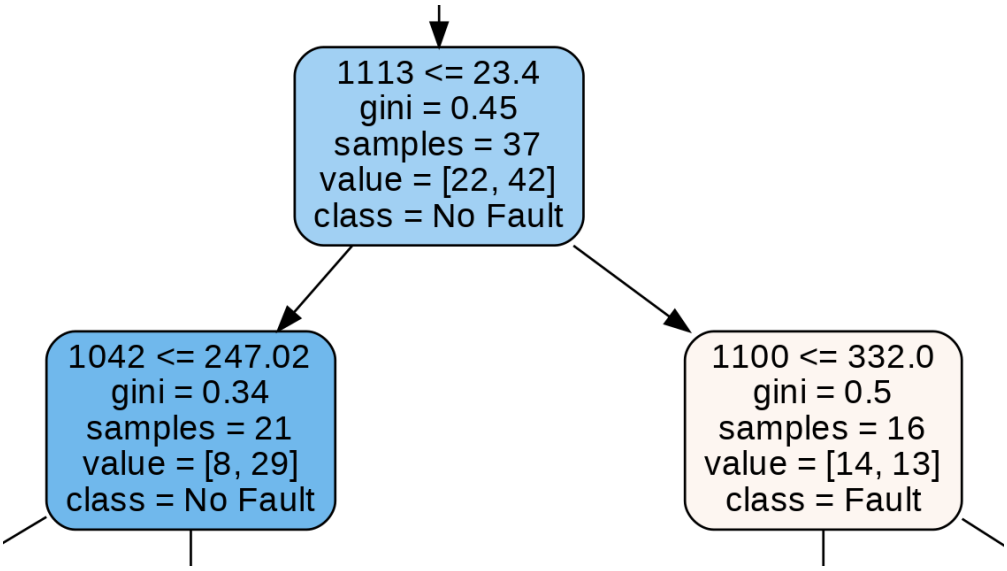

Figure 4.5. An example of split in a classification tree

The example is taken from the data we worked on and shows a node in a tree along with its children nodes. In each node, the split condition on the vehicle parameter (represented by a numeric code), the Gini index and the corresponding predicted class label are shown. Thus, a path from the root to a leaf is just a sequence of IF-ELSE propositions chained with AND conditions. We will call it a **rule**.

The RF technique often generates models with high generalization capabilities and has the advantage to be rather fast to fit and to produce inference from. Nevertheless, even if the base learner is a clearly interpretable model (in fact, certainly one of the most intuitively comprehensible models among the most popular ones), the resulting ensemble model is a black box. In fact, hundreds or even thousands of base models, although individually interpretable, are not easily inspectable in practice. The algorithm discussed in this section aims, in its original implementation, at extracting a minimal subset of rules that mimics as closely as possible the RF performances on a given dataset, thus being

a distilled and interpretable version of it. The latter was proposed in [5]. We applied some modifications (which we will discuss) aimed at obtaining a better suited procedure to our specific task and repurpose the algorithm to the extraction of rules describing the *anomaly* class of a supervised dataset with binary labels, on which the RF is built. For what concerns the technical details, we chose Python language as the coding language and conducted tests on Google Colab first and eventually switched to a multi-core Microsoft Azure Virtual Machine.
In general terms, the algorithm takes a supervised dataset (having binary labels obtained by an anomaly detection phase) as input and performs the following steps:

1. Fit a Random Forest on the dataset
2. Break the RF apart, dividing it in all the rules that it is composed of
3. Assign a score to each rule
4. For a fixed number of random restarts:
    - Start from an initial random set of rules and explore the RF rules space (with some heuristic, exhaustive search is clearly infeasible) to find a (sub-)optimal rule set that optimizes some performance measure
5. Test the optimal rule set on a test sample

In the next paragraphs, we will describe in detail the various phases of the algorithm, as it is presented in the original paper.

Inputs definition

The algorithm requires the following inputs:

- `trainSet`, `testSet`: input data split in training and test set
- `iniRuleNo`: the number of rules included in the initial random set at each random restart of Stochastic Hill Climbing

- `treeNo`: the number of trees in the Random Forest

Building of the Random Forest

A Random Forest `RS` of `treeNo` trees is fit on `trainSet`. Then, the set `Rs` of all terminal nodes of the trees composing the forest is obtained. Recall that each terminal node corresponds to a classification rule, so, in practice, this phase extracts all the rules employed by the RF to classify samples.

Computing the rules coverage

Let us define `m = size(trainSet)` and `n = size(Rs)`, respectively the number of samples in the training set and the total number of rules of the Random Forest. A sparse matrix, initialized with all zeros, `RsCoverage = zeros(m,n)` is built and, for each sample in `trainSet` and for each rule in `Rs`, if the rule matches the sample, then the algorithm assigns `RsCoverage(sample, rule) = class`, where `class` is the class prediction of `rule`. A rule matching a sample just means that the latter's features pass all the conditions the rule is composed of. Then, `RsCoverage` matrix just shows which rule covers each sample and the corresponding class prediction.
This matrix is then used to assign a score to each rule, which in turn will then control the rule selection process. Various score functions may be designed, depending on which kind of rules we want the algorithm to search for. The authors of the paper propose the following score function:

$$ruleScore_1 = \frac{cc - ic}{cc + ic} + \frac{cc}{ic + k} \tag{4.25}$$

where

- cc = number of training samples covered by the rule and correctly classified
- ic = number of training samples covered by the rule and incorrectly classified

- k = predefined positive constant, which is introduced just to avoid the division by zero in the second fraction

The function defined in (4.25) is designed to retain rules with high classification accuracy and high coverage. Clearly, other measures of rule performance may be designed and tested. As an example, the authors propose a second score function:

$$ruleScore_2 = ruleScore_1 + \frac{cc}{rl} \tag{4.26}$$

where rl is the rule length (*i.e.*, the number of conditions tested by the rule). This modification aims at extracting rules with the same characteristics as the first one, while also preferring shorter rules, thus improving interpretability.

Heuristic search in the rules space with random restarts

A selection method is used to generate an initial set of rules `iniRs` (composed by `iniRuleNo` rules) to start searching from. The probability to select a rule is proportional to its score. Then, the heuristic search algorithm is started from `iniRs` and is given a predefined maximum number of iterations `maxIterations` to improve an objective function. This step is repeated `numRandomRestarts` times, each time with a new initial random set of rules and, at the end of this procedure, the best rule set found by random restarts is selected. Also in this case, various different choices are possible for the heuristic search method, as for the objective function. The authors propose Stochastic Hill Climbing as the search method and the mean rule accuracy (computed on the current rule set) as the objective function. Note that it is not totally obvious how a neighbour of a rule set in the RF rule space should be defined. In this case, it is assumed that the neighbours of a given rule set S are:

- all sets obtained from S by adding a rule
- all sets obtained from S by removing a rule

To add rules, the same selection procedure used to build `iniRs` is used, while, when removing one, the probability to remove the rule is inversely proportional to its score (so that lower scored rules have higher probability to be removed).

Assessing performance on test set

Finally, the rule set obtained is tested on `testSet`, where various measures of performance may be computed. The authors of the paper do not specify how the prediction on a new sample is made from the rule set. Thus, we implemented a `testRuleSet` function, which has the following pseudocode:

```
ruleSetPredictions = testRuleSet(ruleSet, samples):
  predictionsMatrix = buildPredictionMatrix(ruleSet, samples)

  ruleSetPredictions = emptyList()
  numRules = size(ruleSet)
  numSamples = size(samples)

  for each sample in samples:
    notCovers = 0
    notFault = 0
    fault = 0

    for each rule in ruleSet:
      pred = rule.predict(sample)

      if pred == -1: #the rule does not cover the sample
        add 1 to notCovers
      elif pred == 0: #the rule classifies the sample as normal
```

```
        add 1 to notFault
      else: #the rule classifies the sample as anomalous
        add 1 to fault

    if notCovers == numRules:
      append majorityClass to ruleSetPredictions
    elif notFault == fault:
      append majorityClass to ruleSetPredictions
    else:
      perform majority vote among rules of ruleSet
      append majorityVote to ruleSetPredictions

return ruleSetPredictions
```

The function builds a `predictionsMatrix`, in the same way the `RsCoverage` matrix is built. Then, for each sample, it computes the number of rules which do not cover the sample and the number of rules that classify the sample respectively as normal or anomalous. Finally, the different possible outcomes are managed: if no rule covers the sample or a tie occurs, the majority class is predicted from the rule set. Otherwise, a majority vote is taken.

Computational time and memory issues

Some steps in the algorithm require considerable computational effort, which can lead to extremely long calculation times or exhaustion of the runtime memory. In particular, the construction phase of the `RsCoverage` matrix is characterised by both these difficulties at the same time. In order to have some reference measures (remembering that such a matrix has a number of rows equal to the number of samples and a number of columns

equal to the number of extracted rules), the reader should consider that our dataset was composed of about 35000 elements, while the total number of extracted rules is in the order of tens/hundreds of thousands, depending on the algorithm parameters.
Therefore, after an initial standard implementation, it became necessary to handle these issues more carefully. As far as memory management is concerned, large matrices are built using sparse matrices (exploiting SciPy library) and through an iterative process, a certain number of columns at a time. In fact, in the first implementation, the construction of the lists containing the values and row and column indices required to define the sparse matrix led to the memory of the machine used being exhausted.
On the other hand, calculation times have been greatly improved by the parallelization of the most computationally demanding phases of the algorithm. This task was accomplished using *Ray* (https://www.ray.io), an open source Python package that allows to scale computational intensive workload with minimal modifications of the original code.

4.2.2 Ad-hoc modifications

The original algorithm was designed solely with the objective of yielding an interpretable version of a Random Forest model. However, the objective of our project was slightly different, or at least not limited to the interpretability of the model. In fact, we are interested in obtaining rules that predict (and thus characterise) the anomalous class, in order to use these rules as an input for an in-depth analysis of the system's fault conditions. However, the strong unbalance in labels provokes a pronounced tendency of the Random Forest to have much more rules predicting normal samples than rules predicting anomalous samples. In addition, these rules also tend to have higher scores than the rules predicting faults, because they cover more samples and usually have higher accuracy. This is due to the higher frequency of normal samples and it is not necessarily related to their quality. Thus, it's easy to end up having a rule set almost exclusively containing rules which are not useful to the analysis of faults. This is an issue that is not limited to Random Forests, but it is typical of strongly unbalanced data. In fact, any machine learning algorithm will

tend to learn more from normal samples simply because it has much more experience on them than on anomalous samples and, also, performance measures are improved most by learning from normal than anomalous samples. For this reasons, we applied the following adjustments to the algorithm:

- Only rules predicting the anomalous class are extracted from the Random Forest
- We tested several objective functions for the heuristic search phase and different ways to attribute scores to rules in order to force the algorithm to extract the rules that are best in predicting faults

In particular, the first change, combined with the way we implemented the prediction computation from sets of rules, means that a rule set predicts

- *Fault* if at least one rule in the rule set covers the sample (and thus predicts the *Fault* class)
- *Normal* if no rule in the rule set covers the sample

To validate the benefit caused by this modification, we executed the rule extraction algorithm on a small Random Forest of 20 trees, both extracting all rules and extracting only the rules which predict *Fault*. We also reduced the iterations of the heuristic search, in order to speed up the computation. The obtained rules were then applied on a test set. Regarding the labelling of the first type, the RF achieves the performances described in Table 4.1, while Table 4.2 refers to the extracted rule set without our modification and Table 4.3 refers to the extracted rule set made only of rules predicting the *Fault* class. Regarding the measures employed, we recall that, in terms of true/false positives/negatives:

- *Accuracy* is the percentage of correct predictions, considering both classes

$$accuracy = \frac{tp + tn}{tp + tn + fp + fn} \tag{4.27}$$

- *Precision*, as in 3.1
- *Recall*, as in 3.2

Random Forest performance	
Accuracy	92.56%
Precision	84.5%
Recall	26.83%

Table 4.1. Performance of small RF on data with first logic labels

Rule Set performance	
Accuracy	90.7%
Precision	72.73%
Recall	3.81%

Table 4.2. Performance of rule set extracted from small RF on data with first logic labels, without removing the rules predicting normal samples

The values shown in the tables suggest the benefit of our proposal: without it, the extracted rule set has an extremely low recall on the *Fault* class, even though it retains an accuracy comparable to that of the original model. With our modification, however, the recall is also preserved to a satisfactory extent.

The results for labelling of the second type further confirm the above. The RF performance is shown in Table 4.4, while rule sets performances with and without our modification are shown respectively in Table 4.5 and 4.6. Without removing rules that predict normal samples, the algorithm is not even able to extract rules characterizing anomalies and thus never predicts the *Fault* class.

Random Forest performance	
Accuracy	99.50%
Precision	100%
Recall	17.95%

Table 4.4. Performance of small RF on data with second logic labels

Rule Set performance	
Accuracy	91.91%
Precision	77.14%
Recall	21.43%

Table 4.3. Performance of rule set extracted from small RF on data with first logic labels, removing the rules predicting normal samples

Rule Set performance	
Accuracy	99.40%
Precision	0%
Recall	0%

Table 4.5. Performance of rule set extracted from small RF on data with second logic labels, without removing the rules predicting normal samples

Rule Set performance	
Accuracy	99.24%
Precision	39.13%
Recall	46.15%

Table 4.6. Performance of rule set extracted from small RF on data with second logic labels, removing the rules predicting normal samples

4.2.3 Alternative scoring and objective functions

In 4.2.1 we mentioned the rule scoring function and the heuristic search objective function originally considered by the authors of the algorithm. We describe some variations which, in some cases, perform better on the specific problem at hand.

Rule scoring functions

The original rule score does not take into account the number of samples that a rule was not able to classify, *e.g.* the samples for which the conditions of the rule were not satisfied.

We propose the following modification

$$ruleScoreModified = \frac{cc - ic}{cc + ic} + \frac{cc}{ic + k} + \frac{cc}{nc + k} \tag{4.28}$$

where nc is the number of anomalous samples (*i.e.* samples with label 1) that the rule did not apply to. The additional term is meant to attribute a higher score to rules that apply to a higher number of examples of fault.

Heuristic search objective functions

Several alternatives may be considered as the objective function that the Hill Climbing algorithm tries to improve. Beyond the authors' idea, we propose two performance measures based on the rule set as a whole, rather than on the single rules it is composed of. These are computed retrieving, from the `RsCoverage` matrix, the predictions of the current rule set members on the training set and then obtaining the corresponding predictions of the entire rule set. From these, standard performance measures may be computed and become the objective to be optimized:

- **Mean accuracy**: objective function proposed in the original paper. Computes the accuracy of each rule in a rule set

$$ruleAccuracy = \frac{cc}{cc + ic} \tag{4.29}$$

  and then averages them on the rule set.

- **Rule set precision**: precision on the *Fault* class of the rule set, as in 3.1.

- **Rule set recall**: recall on the *Fault* class of the rule set, as in 3.2.

4.3 Results

In this section, we present and analyse the results obtained with the rule extraction algorithm. First, for each labelling logic, we establish the Random Forest results on the test sample, which stand as a baseline. Then, the performance of the extracted set of rules on the same test sample is described and compared with the original model.
The latter is a Random Forest Classifier featuring the following parameters (established through a cross-validated grid search) for the first labelling logic:

- *Number of trees*: 50
- *Split quality measure*: Gini impurity
- *Maximum depth of trees*: 15
- *Minimum number of samples required to split an internal node*: 2
- *Minimum number of samples required to be at a leaf node*: 1

and for the second one:

- *Number of trees*: 500
- *Split quality measure*: Gini impurity
- *Maximum depth of trees*: 47 (*i.e.* the number of features in the dataset)
- *Minimum number of samples required to split an internal node*: 2
- *Minimum number of samples required to be at a leaf node*: 1

while the Stochastic Hill Climbing algorithm has the following default parameters:

- *Number of iterations for each restart*: 500
- *Number of random restarts*: 10

- *Number of rules in the initial random set*: 50

As discussed above, we enabled the filtering out (before heuristic search) of rules predicting normal samples in both cases.

4.3.1 First labelling logic

Dataset characteristics

The employed dataset has 33053 rows and 49 columns. The distribution of samples between classes is: 29905 *Normal*, 3148 *Fault*, as Figure 4.6 shows.

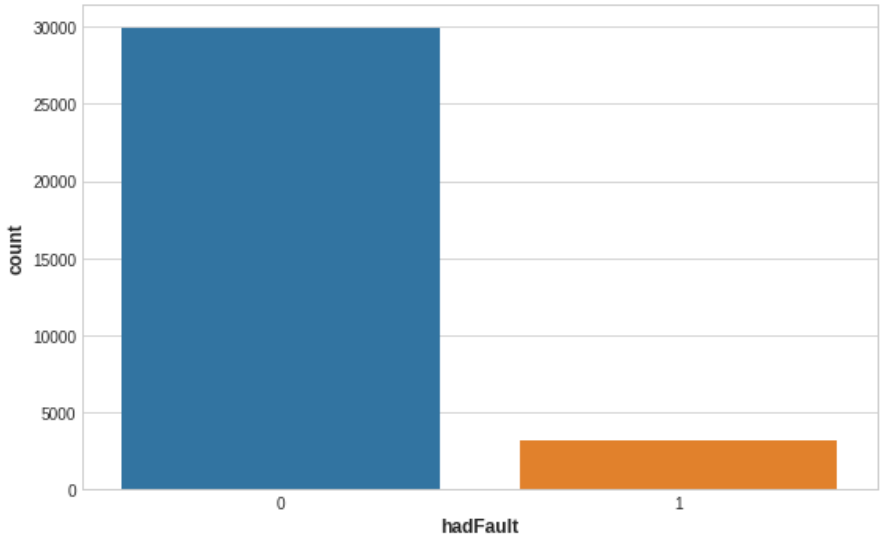

Figure 4.6. The distribution of classes of the first dataset

Random forest baseline

The RF is shown in Table 4.7. Accuracy is quite high and the fault prediction is fairly reliable, detecting about one fault in every tree with about 20% of error rate.

Random Forest performance	
Accuracy	92.72%
Precision	80.41%
Recall	31.27%

Table 4.7. Performance of Random Forest Classifier on data with second logic labels

Rule extraction

The rule scores were attributed by means of the $ruleScoreModified$ function, while heuristic search in the RF rule space with Stochastic Hill Climbing was ran testing all three alternative objective functions presented in 4.2.3. Results are shown in Table 4.8.

		Objective function		
		Mean rule accuracy	Rule set precision	Rule set recall
Measure	Accuracy	91.23%	91%	90.17%
	Precision	58.93%	56.41%	48.36%
	Recall	26.19%	24.44%	46.83%

Table 4.8. Results of extracted rule sets from RF fitted on dataset with first logic labels

	Objective function		
	Mean rule accuracy	Rule set precision	Rule set recall
Initial # rules	20015	20015	20015
Final # rules	329	296	438
RR improvements	3	3	2

Table 4.9. Characteristics of the extracted rule sets from RF fitted on dataset with first logic labels

Table 4.9 shows the total number of rules (which predict *Fault*) of the original RF, the number of rules in the final rule set and the number of improvements obtained through random restarts. Mean rule accuracy seems to be the most appropriate objective function for this type of labelling.

4.3.2 Second labelling logic

Dataset characteristics

The employed dataset has 32265 rows and 49 columns. The distribution of samples between classes is: 32068 *Normal*, 197 *Fault*, as Figure 4.7 shows.

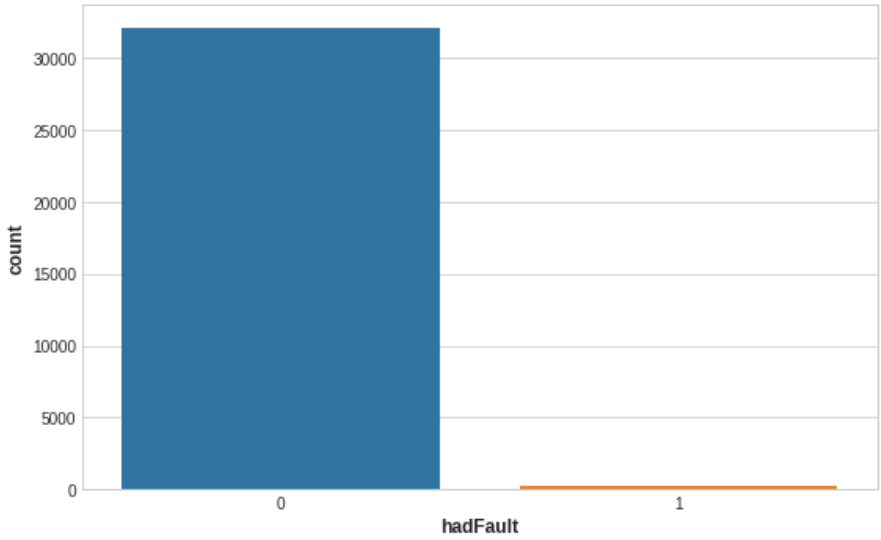

Figure 4.7. The distribution of classes of the second dataset

Random forest baseline

The RF baseline performance is shown in Table 4.10. The model has an almost perfect accuracy and perfect precision, being totally reliable when predicting a fault. Also, it detects about one fault in every three.

Random Forest performance	
Accuracy	99.58%
Precision	100%
Recall	30.77%

Table 4.10. Performance of Random Forest Classifier on data with second logic labels

Rule extraction

The rule scores were attributed by means of the $ruleScoreModified$ function, while heuristic search in the RF rule space with Stochastic Hill Climbing was ran testing all three alternative objective functions presented in 4.2.3. Results are shown in Table 4.11.

		Objective function		
		Mean rule accuracy	Rule set precision	Rule set recall
Measure	Accuracy	98.83%	99.32%	99.41%
	Precision	26.92%	44.68%	51.02%
	Recall	53.85%	53.85%	64.10%

Table 4.11. Results of extracted rule sets from RF fitted on dataset with second logic labels

	Objective function		
	Mean rule accuracy	Rule set precision	Rule set recall
Initial # rules	27854	27854	27854
Final # rules	179	101	84
RR improvements	3	2	1

Table 4.12. Characteristics of the extracted rule sets from RF fitted on dataset with second logic labels

Table 4.12 shows the total number of rules (which predict *Fault*) of the original RF, the number of rules in the final rule set and the number of improvements obtained through random restarts. Overall, the recall of the rule set on the anomalous class seems to be the most effective objective function for the heuristic search phase. In fact, the rule set obtained has the highest performance measures with the lowest number of rules. However, the extracted rule set has halved the precision of the RF and doubled its recall, meaning it is much more imprecise.

Some examples of the rules generated through the latter as objective function are presented below (full names of parameters have been obscured for privacy):

```
Rule instance contains the following tests:
```

```
- parameter 1029 > 41.734375
- parameter 1106 > 277.78125
- parameter 1018 > 87.328125
- parameter 1100 > 1012.0
- parameter 1056 > 55.39999961853027
- parameter 1009 > 1.0
- parameter 1106 <= 290.421875
- parameter 1055 > 22.5
=> Fault


Rule instance contains the following tests:
- parameter 1042 <= 294.53125
- parameter 1229 > 4.8046875
- parameter 1057 > 2.7705078125
=> Fault


Rule instance contains the following tests:
- parameter 1029 > 24.1875
- parameter 1108 > 147.03125
- parameter 1058 <= 1.2119140625
- parameter 1029 <= 24.640625
- parameter 1057 <= 2.8740234375
- parameter 1034 > 23.5
- parameter 1019 <= 39.5
- parameter 1018 > 45.796875
- parameter 1106 <= 341.328125
```

```
- parameter 1108 <= 274.078125
- parameter 1056 <= 99.39999771118164
- parameter 1229 <= 4.08984375
- parameter 1034 <= 26.5
- parameter 1018 <= 81.484375
- parameter 1028 <= 36.5
- parameter 1034 <= 30.5
- parameter 1108 <= 310.453125
=> Fault
```

4.4 Discussion of results and possible improvements

The proposed rule extraction method manages to obtain a set of rules which replicate the original Random Forest performance fairly well, considering that the number of rules employed is several orders of magnitude smaller than the total number of rules of the forest. The reduction of the number of rules is higher for the second dataset, but this is probably due to the fact that, because the latter's labels individuate a much smaller group of faults, less rules are needed to characterize all considered faults and replicate the RF's performances. For the first dataset, much more faults are considered, thus much more rules are needed. We also note that, in both cases, the recall of the rule set on the *Fault* class was the objective function that led to less improvements through random restarts. This may suggest that, for this particular function, the Stochastic Hill Climbing should be given more iterations and more random restarts to better optimize the objective.
In any case, the performance degradation is notable and the extraction process could certainly be improved. Obviously, other functions to attribute scores to rules and to be used as objective functions certainly may represent a potential direction of research. However, the performance bottleneck of the whole process is, more probably, located in the phase of heuristic search, which exploits a very simple and inefficient optimization

algorithm. Thus, solutions more close to the state-of-the-art should be implemented and tested (such as simulated annealing or genetic algorithms) and would certainly provide considerable enhancements to the algorithm performances.

Chapter 5

Conclusions and Future Work

The results presented in 4.3 represent just one side of the evaluation of the proposed method. In fact, they are just related to the ability of the extracted rule set to replicate the classification performance of the original RF fitted on the given labelled dataset. However, a full evaluation of the proposed method cannot be complete without the judgement and domain knowledge of the client firm's professionals and, possibly, some validation tests conducted directly on the involved vehicles. In particular, it should be verified if the kind of fault conditions identified by the extracted rules correspond to the actual issues on the engines and to what extent. From this point of view, a proper validation of the extracted sets of rules has not been carried out yet. However, the results obtained have been found satisfactory by the client's firm professionals and a follow up project has received funding from the client company.
Therefore, possible directions of research aimed at perfecting the algorithm are certainly represented by

- Increasing the effectiveness in the heuristic search phase through the implementation of state-of-the-art solutions for this kind of problems, rather than the current simple optimization algorithm

- Validating the extracted rules through on site tests, in order to evaluate the extent to which they manage to characterize fault/anomalous conditions

Beyond the specific application considered in this work, the contribution of this thesis consists in providing a general method to explain and characterise anomalies found by an anomaly detection method, exploiting the known reliability of Random Forest Classifiers. The complete **detection-explanation** pipeline takes as input an unsupervised dataset and proceeds as follows:

1. Anomaly *detection*: anomalies are identified by a suitable statistical method applied on the initial unsupervised dataset, such as one of the algorithms proposed in Chapter 3 or some other more specific procedure. This phase yields a supervised dataset with binary labels, which represent normal and anomalous examples

2. Anomaly *explanation*: apply the Rule Extraction algorithm to the supervised dataset to obtain rules which characterise the anomalous class, thus being an explanation of the latter

As shown in 4.3, the algorithm shows promising results both in cases in which multiple types of different anomalies are present and in cases in which a more circumscribed anomaly has to be characterised.
In conclusion, in the future we will improve the algorithm in the weak points described above and test the whole pipeline in other cases with different datasets.

Bibliography

[1] Aurelien Geron. *Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, Sebastopol, CA, 2017. ISBN 978-1491962299.

[2] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

[3] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning: with Applications in R*. Springer, 2013. URL https://faculty.marshall.usc.edu/gareth-james/ISL/.

[4] Marco Longo. Design and development of a fault monitoring system for automotive vehicles, 2020.

[5] Morteza Mashayekhi and Robin Gras. Rule extraction from random forest: the rf+hc methods. 06 2015. ISBN 978-3-319-18355-8. doi: 10.1007/978-3-319-18356-5_20.

[6] Kishan G Mehrotra, Chilukuri K Mohan, and HuaMing Huang. *Anomaly detection principles and algorithms*. Springer, 2017.

[7] Okpedia: enciclopedia online di economia e tecnologia. Local search, 2021. URL https://www.okpedia.it/ricerca_locale. [Online; accessed October 13, 2021].

[8] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3 edition, 2010.

[9] Wikipedia, the free encyclopedia. Anomaly detection, 2021. URL https://en.wikipedia.org/wiki/Anomaly_detection. [Online; accesssed October 21, 2021].

[10] Wikipedia, the free encyclopedia. Kernel regression, 2021. URL https://en.wikipedia.org/wiki/Kernel_density_estimation. [Online; accesssed November 04, 2021].