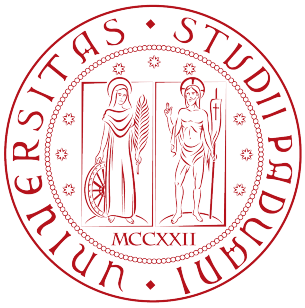


UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI FISICA E ASTRONOMIA "GALILEO GALILEI"

Master Degree in Physics of Data

Final Dissertation

Industrial application of Machine Learning: Predictive Maintenance for failure detection

Thesis supervisor
Dr. Jacopo Pazzini

Thesis co-supervisor
Stefano Campese

Candidate
Federico Agostini

Academic Year 2020-2021

Contents

List of Figures

List of Tables

1 Introduction

In 2011, the term "Industry 4.0" was used for the first time in the "High-Tech Strategy" project presented by the German government to indicate "the comprehensive transformation of the whole sphere of industrial production through the merging of digital technology and the internet with conventional industry". After that, this word started to be applied to describe the fourth industrial revolution, which aims to define a stronger and stronger integration between physical and digital systems. New and innovative technologies guide this process: internet of things, cybersecurity, big data, machine learning, cloud computing, autonomous systems.

1.1 Industrial Maintenance in the Machine Learning era

Industry 4.0 is characterized by cyber-physical systems which embed sensors to collect and transmit data. As a consequence, the amount of information extracted during the production processes has increased exponentially [1]. This knowledge represents a key factor for companies if properly analyzed and understood. Interpreting these data can provide advantages such as maintenance cost reduction, fault reduction, increase productivity, and more.

Equipment maintenance is a fundamental aspect in industries since it affects resource lifetime and efficiency. Furthermore, failures imply shutdowns in production processes in addition to the cost of repairing or replacing the systems. Maintenance can be classified as follows [2]:

- Run-to-Failure (R2F) consists of maintenance interventions only when the equipment stops working. This is the simplest approach but implies the summation of repair costs with a production stop. Furthermore, standstill can last for long time frames if pieces to replace are not immediately available. On the other hand, stocking spare parts implies additional charges.
- Preventive Maintenance (PvM) is based on planned schedules, where interventions occur based on time or process iterations. This approach usually avoids failures, but also leads to inefficient use of resources since actions are taken even if unnecessary. In fact, parts are substituted on the basis of historical information or details from the constructor, without asserting their effective wear. The consequence is that still working pieces are replaced even if they do not reach the end of their life cycle.
- Predictive Maintenance (PdM) provides an estimate of systems health status. It requires continuous monitoring of the equipment but allows early detection of failures tanks to predictive tools based on historical data, integrity factors, statistical inference methods, and engineering approaches.

R2F delays maintenance actions as long as possible, with the risk of unavailability of the equipment and a slow down (if not a shutdown) of the entire production chain. On the opposite side, PvM anticipates interventions but has the risk to replace parts that are still not close to the end of their life cycle. PdM stands in the middle, reducing equipment failures and while minimizing maintenance

costs. This is achieved by evaluating the system health state in real-time, in order to estimate the next failure. The task is challenging since it requires collecting data constantly from sensors on the machines and process them to provide a view of the state of the apparatus. A good approach needs also to be responsive to sudden changes in the conditions, react immediately, and supply information about possible failures. Thus, PdM allows using the whole life cycle of machine parts, while avoiding interruptions due to unexpected faults.

Machine learning is becoming a fundamental tool to develop intelligent algorithms that can diagnose failures. This is due to the ability to handle high-dimensional and multivariate data and the ability to extract relationships between them in complex scenarios, even where they are not trivial or hidden behind correlations between multiple factors. In addition, they can be applied even without having a mechanistic knowledge of the underlying processes. Performances however heavily depend on the choice of the ML technique, which in turn highly depends on the production environment and the available variables.

1.1.1 Predictive Maintenance in the Literature

Machine Learning provides an extensive amount of different techniques to tackle a large variety of tasks. In the context of PdM, the most implemented algorithms include Random Forest and Neural Networks approaches (including fully connected networks, convolutional, long-short term memory, and other deep implementation), followed by Support Vector Machines and k-means. The study described in [1] also highlights that each work uses specific equipment, such as turbines, motors, compressors, pumps, fans, and others.

Random Forest is the most used one since decision trees allow a large number of observations to be part of the model; in addition, they reduce variation and increase generalization.

Neural Networks do not require any expert knowledge about the specific industry field and could work well even if data are inconsistent; they are able to extract correlations between variables without modeling them a priori. However, they require large amounts of observations, which implies building solid pipelines to acquire and manage sensor data that monitor equipment status. In addition, their conclusions may results difficult to interpret.

Support Vector Machines are widely used to perform regression and classification tasks. Their ability to perform accurate classification even in multi-class scenarios is particularly exploited for industrial applications. Disadvantages comprehend the difficulty to choose an appropriate kernel function, the training time that does not scale well with the amount of data, and the difficulty to understand the model and incorporating business logic.

k-means is applied to find the clusters of a dataset. It is easy to implement, has good performance even with lots of samples and tends to minimize inter-class variance, and increases the extra class distance. The major downside is represented by the difficulty to determine the correct number of clusters k, but some algorithms can provide reliable estimates of this parameter even without any prior knowledge.

Beyond the specific choice of algorithm, each PdM case is unique and requires a specif implementation that depends both on the application and the variables available (which in turn are subject to the environment and the equipment). Any change to some of these variables may result in a totally different approach to tackle the task. On top of that, there is no large and well-established public dataset. As consequence, it is difficult to compare performances of different solutions, due to the lack of universally acknowledged benchmarks; this aspect is in contrast with other ML fields such as image processing, where lots of publicly available data can be used (MNIST and ImageNet as examples).

1.2 Case study of this work

This work analyzes data coming from a leader company in the field of refrigeration systems. Their machines feature a series of sensors that monitor mechanical and thermodynamics variables; these trigger some encoded alarms which are the actual data collected by the infrastructure. The processes

that associate physical variables (temperature, pressure, ...) to alarms are unknown. Eventually, alarm counts are registered along with information about the time and identifiers of the machines. Data are collected from more than 2000 centers, and each of them gathers a few dozens of different refrigerating machines.

Alarms are classified into about 80 categories by their ID flag; the correlation between this identifier and the physical state of the machine is unknown. That is, it is not possible for example to associate a certain alarm A with issues related to temperature, pressure, or even combinations of different variables. Ids are also associated with a descriptor indicating their severity. Again, the motivations behind this assignment are not made available by the company. The absence of insight into these correlations makes it impossible to build some hypothesis a priori. Hence, all the details must be extracted from the data.

Two more datasets contain information about requests for technical support and reports regarding the repair interventions. These documents are very noisy and it is quite difficult to extract relevant details without deteriorating the quality of the data. For example, technical support requests contain both machine failure notifications (which is fundamental for developing the predictive maintenance algorithms) and other events such as baseboard or light substitutions, which are completely uncorrelated to alarm signals. On the other hand, repair reports present lots of compilation errors. Dates annotated in them are not strictly connected to a specific alarm or support request; often reports are written only several days (if not months) after the conclusion and they may include more technician operations in the same ticket. These traits cause high difficulty to extract meaningful information, which in turn determines the performance of the PdM solution since failures are withdrawn from those reports.

Summarizing, the dataset is hence composed of a multivariate time series of alarm counts for each of the different centers; no physical variables are available. This information needs to be joined with repair requests, in order to produce a supervised setting where failures may be predicted by an algorithm. In addition to that, a Natural Language Processing (NLP) analysis can be performed using the other two data collections in order to extract meaningful reports in an automated way.

1.3 Dataset Description

As stated in the previous section, variables come from the logs sent from the operating machines in real-time. They are stored in an Amazon S3 bucket and can be downloaded through a SQL query to the database. Data are "raw", in the sense that they represent exactly the signals sent by the devices. No filtering nor cleaning procedure is implemented when storing records. The consequence is that the dataset contains a huge amount of noise, and needs to be prepared in order to perform any analysis.

From the queries to the database, three main files are generated and saved in `csv` format.

***"Alarm records"* dataset** collects the alarm signals. It contains about 50.5 million entries. Each row represents a single trigger, described by 32 different fields. They comprehend registry information, localization codes, flags to identify single machines, codex to map together different elements inside the company database, ... The most informative elements among them are:

- date and time (`YYYY-MM-DD HH:MM:SS`) of the alarm;
- codex to locate the facility where the machine is placed;
- code of the report signal, which contains a sub-string useful to discriminate alarms related to scheduled maintenance;
- numeric identifier of the specific alarm.

In addition to that, each ID can be associated with a severity flag through a dictionary, which defines four different levels.

***"Operations"* dataset** groups the reports after telematic or in situ operations are performed due to failures. It consists on about 500 thousand events, each one expressed by 60 features. Each row represents an issue ticket, which is opened when failures are communicated to the main company center. After that, the procedure implies first telematic support, followed by technicians operations in loco if the remote attempt does not fix the problem. As previously stated in the case of *Alarm records*, most columns display information related to registry entries which are used by the company systems to track all the records. Among the useful features for this work, there are:

- date and time notes about the issue ticket opening and its conclusion;
- text entries which are the help request emails sent to the central office to communicate the faults;
- failure type described by a numeric ID assigned after the conclusion of the operation.

***"Assistance calls"* dataset** includes the issue tickets sent to the central office due to malfunctions. There about 630 thousand rows with 26 columns. The structure is similar to the *Operations* dataset. It differs from it since the events are registered as soon as a help request is sent, while in the previous database a maintenance pipeline needs to be opened first. This also explains the difference in the number of events between the two. More than one entry of the *Assistance calls* can be mapped into the same record in *Operations*, since they are merged together in the same maintenance pipeline. In addition, obviously, there is no ID indicating the fault type, since a technician intervention is required to identify the cause of the failure.

In order to deal with these files, some basic preprocessing is required. First of all, each column is manually investigated to extract the data type, since there are some inconsistencies between entries. Secondly, only entries registered from 2017-01-01 are considered; this is necessary since older records were taken with very different acquisition systems, which may not be comparable with newer (and more reliable) ones. In addition to that, events described by some fault ids are removed since both from company indications and previous studies they are recognized as non-related to alarms. Moreover, all the scheduled maintenance operations are removed. Lastly, alarms are grouped together based on their facility location.

A second step requires joining notions from the different sources. In particular, in order to associate alarm counts to maintenance operations, *Alarm records* is merged with the other two datasets based on the day of the record and the geographical position of the facility.

This work is implemented using a distributed environment with the `Dask` [3] library since some operations exceed the memory limit of the available physical machine. The cluster is composed of 30 nodes and 2 GPUs.

1.3.1 Exploratory Data Analysis

Due to the intrinsic complexity of the dataset combined with the lack of a specific description of the nature of the alarms, Exploratory Data Analysis (EDA) is performed to gain some deeper knowledge about the problem domain. EDA consists of inspecting a dataset to summarize its more relevant characteristic. It usually relies on descriptive statistical analysis and visualization methods. This task alone is pretty complex, due to the nature of the data. In fact, records come from more than 2000 different facilities, which deploy disparate models of machines. Each machine has sensors to monitor 80 types of alarms. Reports cover a period of more than three years, starting from 2017.

The information about the single machine is discarded in this work, and the focus is headed towards analyzing the behavior at a facility level. Even with this simplification, EDA is a hard challenge, since it is impossible to visually represent such a huge amount of data. The first variable that needs to be considered is time; alarm counts define time series. For each different facility, a multi-variate time series can be built joining the information about each alarm type. On top of that maintenance operations should be visualized since the goal is to find possible relations between them and the alarms.

As a first approximation, notions about locations are discarded. The underlying assumption is that similar behavior in the alarms leads to similar faults and then to similar maintenance interventions, independently on the location of the refrigerator machine. However, looking at all alarm types together does not bring clear results; in fact, as displayed in Fig. 1.1, there is a great difference in the magnitude between the alarm ids. The heatmap in Fig. 1.1b clearly shows that some records are very frequent (and tend also to have a high daily count), whereas others are pretty much absent. Since there are no indications from the company, it is difficult to decide whether frequent or infrequent variables should be kept or removed from the analysis. In fact, it is possible that recurring alarms are not so dangerous, while the rarest are fatal failures; on the other hand, it may be plausible that unusual records should be connected to sensors present only on a small number of devices or that have been dismissed.

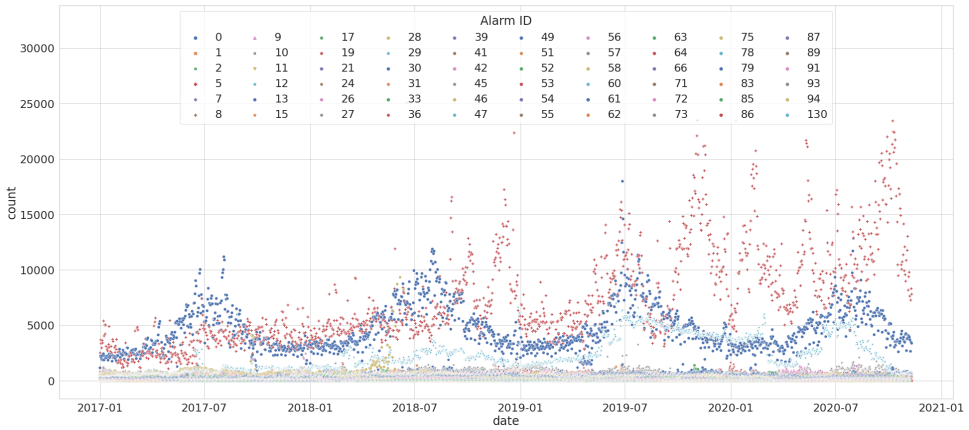
Maintenance runs need to be added into this already complex scenario. To try to visualize them, plots for singular alarm ids are created, still combining data from all the locations. Datasets are filtered by alarm ID; then the alarm counts time series are overlapped by vertical lines, which indicates whether a maintenance operation described by the same alarm ID happened on that day. Results for four variables out of the 80 ids are presented for illustration purposes in Fig. 1.2. Again the setting does not bring great insights about which alarms are more prone to lead to failures. Recurrent ones are usually associated with a high density of maintenance operations, that seems to be independent of the alarm counts. As an example, the frequency of interventions in Fig. 1.2 (visualized by the vertical grey lines density) is not always correlated with sudden increases in alarm counts time series. On the other hand, rare identifiers display also a minor presence of interventions. However, it is not clear if this means that small variations in the variable lead to severe faults, or if they have negligible weight and can be ignored.

Splitting these graphs including the information about the facility where machines are located displays that it is a fundamental variable to consider. In fact, the scenario in Fig. 1.3 clearly indicates that each center has peculiar traits. These include which alarm ids are triggered and their count, along with the number of maintenance interventions. From a visual inspection, it is evident that the alarm count profile does not follow a common trend between facilities; moreover, maintenance runs are actuated in response to different signal events. For example, the profile in Fig. 1.3a and Fig. 1.3c is pretty much constant during the whole period, but there are some isolated spikes in the alarm counts: the major part of the points of the scatter plot lies around the x-axis, but the vertical one spans a large range. Fig. 1.3b shows the presence of two dominant alarms (blue and light blue points), while in Fig. 1.3c all types display a similar magnitude.

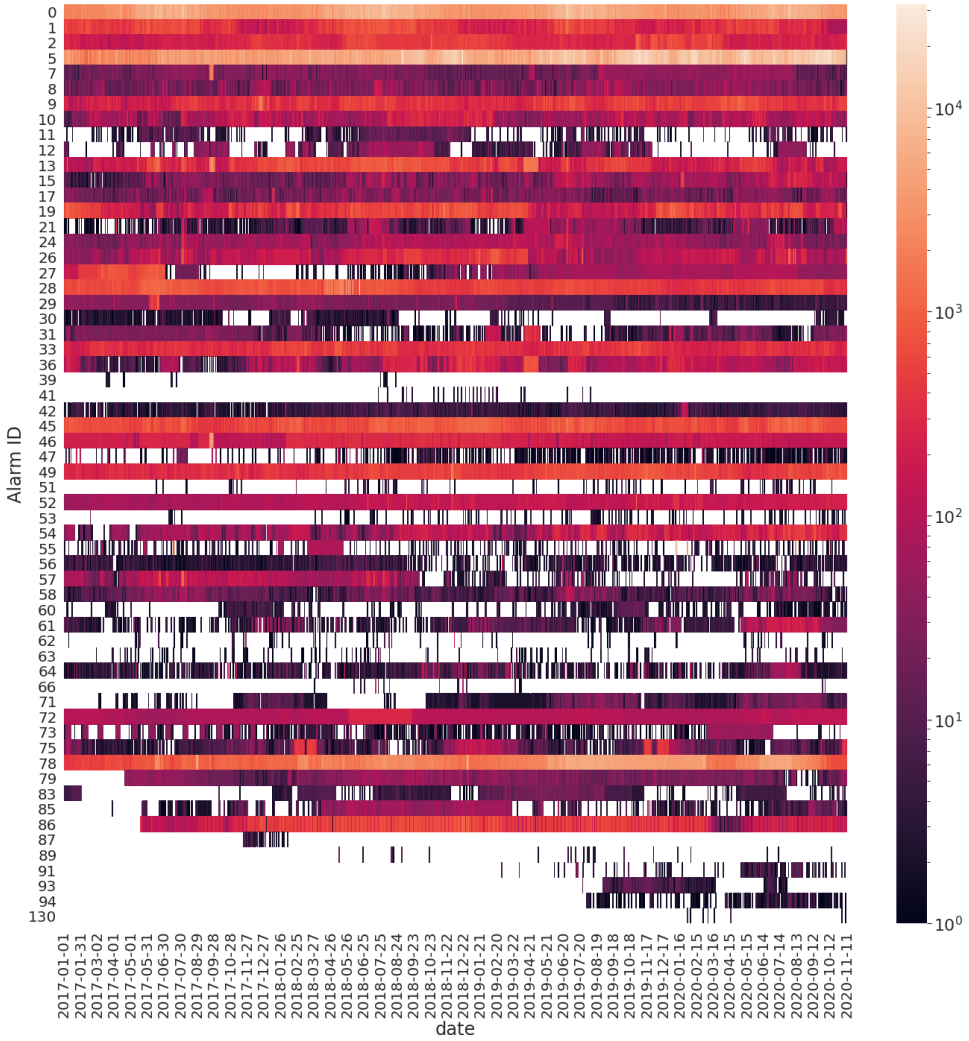
Another aspect analyzed during EDA is the possible correlation between the alarm ids and the label assigned by technicians after maintenance operations (annotated into *Operations* dataset). Again, data are aggregated over all facilities to provide a way to visualize them. In Fig. 1.4, columns refer to the different alarm ids, while rows are the label assigned to the faults. The first noticeable aspect is that a large number of failure types are pretty much never present; the explanation may be that they are old codes that are not used anymore (in fact they are not present in more recent dates). Another interpretation suggests that reports are not compiled carefully, and some labels are incorporated inside other categories. Secondly, each alarm contributes to determining more than one fault, and vice versa, a failure label is not caused by a single warning type. This indicates that breakdowns are due to a combination of different alarms, and cannot be led uniquely back to a specific signal. Correlations between alarms are then present, and it is not immediate to identify them.

Conclusions on EDA Exploratory Data Analysis confirms the complexity of the dataset. The number of different variables which need to be considered is large. Visually, no alarm ids can be selected or discarded from the analysis. In fact, failures appear to be caused by combinations of the different warnings, and the behavior cannot be restricted to a single type. Since correlations are not easy to spot from EDA, they need to be found by algorithms themselves, without any prior knowledge.

Particular importance needs to be assigned to the facility where machines reside. Clearly, each location features unique behavior with respect to others. Distinct alarms are registered with a dissimilar



(a) *Time series representing the daily count of each alarm ID during the observed period. The counts are summed over all the different facilities.*



(b) *The heatmap reports the same information of the scatter plot. Each row represent an alarm type, while the x-axis is the time variable. The color bar ranges over four orders of magnitude and describes the number of daily counts; the logarithmic scale enhances the differences in magnitude.*

Figure 1.1: *Alarm counts grouped by day summed over all the facilities. Both representations show the same information: there is a great variability inter alarm types with respect to their number. Some flags are never registered, while others appear in an exponentially quantity.*

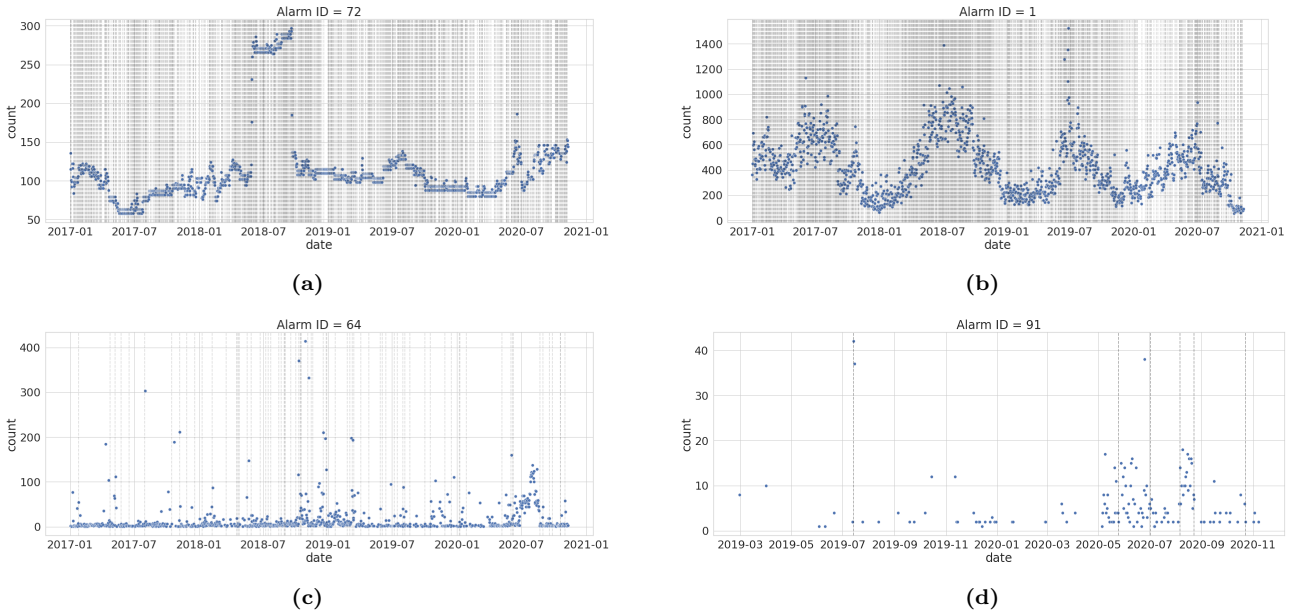


Figure 1.2: *Scatter plots represents the behaviour of 4 different alarms aggregate by day and over the all facilities. Vertical lines show whether a maintenance operation has happened correlated the specific alarm. It can be noticed that there an high variability depending on the ID.*

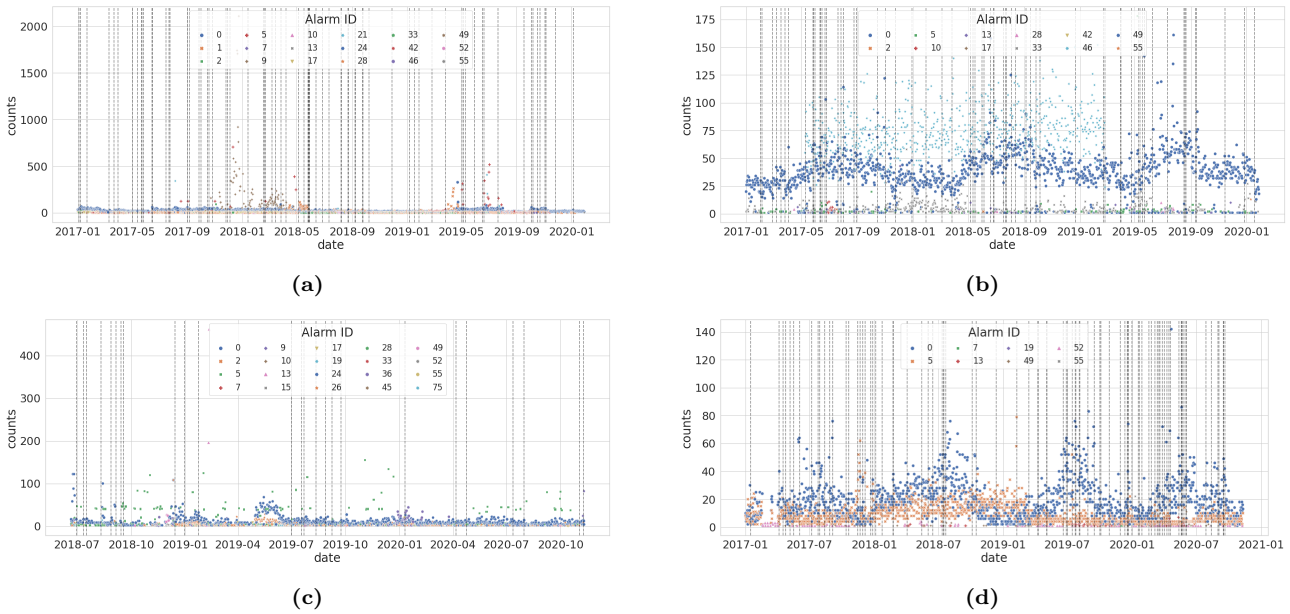


Figure 1.3: *Alarms count (scatter plot) and maintenance operations (vertical lines) for four different facilities. The number of different alarm ID registered varies from center to center, and also their magnitude shows high variability. The density of maintenance interventions is very correlated to the different places, and they are actuated in response to different signals.*

Figure 1.4: *Heatmap which correlates the number of alarms ID for every type of fault identified by maintenance assistance operations. On the horizontal axis there are the different alarm ids, while on the vertical one the label assigned to the failure after the maintenance. The color bar is in logarithmic scale, and it ranges over four orders of magnitude.*

frequency depending on the center. Every single warning has a unique behavior for every specific facility, and a general rule seems not to be present. Even maintenance operations respond to a variety of inputs correlated to the facility. Aggregating data from different places lead to behaviors that do not match the one in single centers. The best solution maybe should be developing a unique model for each specific facility. However, this is unfeasible, since more than 2000 algorithms should be deployed and trained. The solution is to include the location as a variable in the models, and let them find the correlations between alarms time series and the place. The idea is that an algorithm should be able to identify which alarms are more dominant for each facility.

The goal of this work is to build an algorithm to identify failures starting from the alarm counts. Chapter 3 describes the studies applied on a subset of the data. The approach consists of creating multivariate time series collecting observations in time windows of some days in order to predict whether a fault would happen the successive day.

Since results show that a reliable model cannot be built using that subset of data, Chapter 4 focuses on developing an algorithm to select more quality examples. In particular, Natural Language Processing (NLP) solutions are investigated by analyzing the maintenance tickets. Both unsupervised and supervised methods are tested.

To understand the algorithms implemented in the study, Chapter 2 presents the theory underlying these Machine Learning models. In particular, Boosted Decision Trees and Long-Short Term Memory networks are described as applied to predict failures; as for the NLP scenario, unsupervised topic clustering methods are proposed, along with state-of-the-art methods.

As a conclusion, Chapter 5 suggests future studies and developments, along with the proposal of an automated pipeline using the Amazon Web Services (AWS) platform.

2 Theoretical Overview of Machine Learning Algorithms

2.1 Introduction to Machine Learning

Machine Learning (ML) is a field of Artificial Intelligence that aims to develop algorithms that can learn from data automatically [4]. This definition means that the model adapts itself according to the samples it is "trained" on, in contrast to methods that are defined "a priori" before even seen the data.

Focus is headed towards predictions rather than estimation: therefore, we are interested in computer systems that have the ability to forecast new observations without being explicitly programmed.

ML algorithms can be divided into three general categories:

- in supervised learning the algorithm learns from labelled data;
- unsupervised learning involves finding patterns in unlabelled data;
- in reinforcement learning, an agent interacts with the environment and adapts in order to maximize its reward.

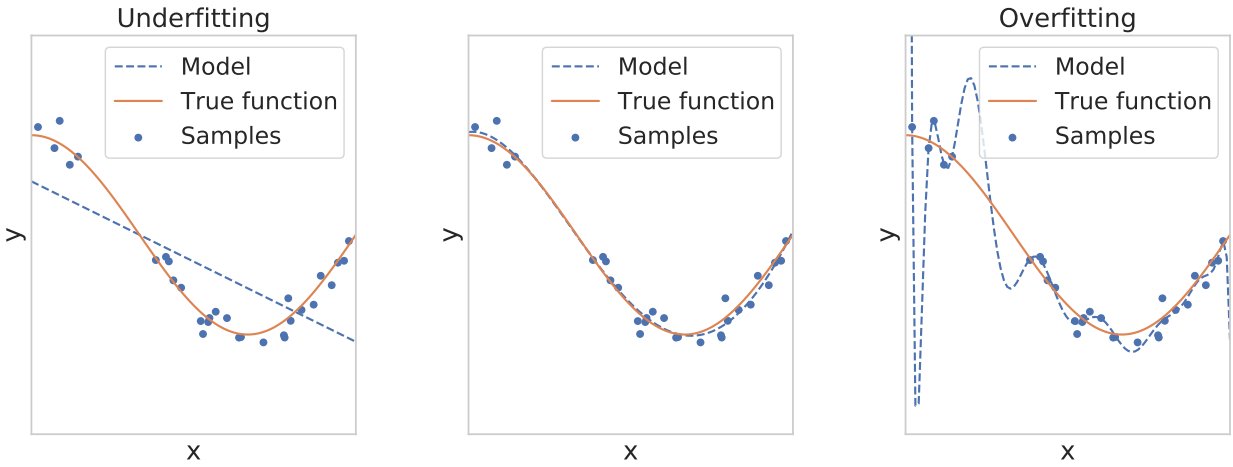
These learning tasks can be described on the basis of some common ingredients:

- the dataset $\mathcal{D}$ is made by a matrix $\boldsymbol{X}$ of independent variables and a vector $\boldsymbol{y}$ of dependent variables;
- the model $f(\boldsymbol{x};\boldsymbol{\theta})$ is a function $f : \boldsymbol{x} \to y$ of the parameters $\boldsymbol{\theta}$ that predicts an output given a vector of input variables;
- the cost function $C(\boldsymbol{y}, f(\boldsymbol{X};\boldsymbol{\theta}))$ allows to evaluate model performances; during the fitting procedure parameters $\boldsymbol{\theta}$ are modified in order to minimize C.

Since the focus of ML algorithms is to generalize the knowledge to unseen examples, the first step in the analysis is to split the dataset into training ($\mathcal{D}_{\text{train}}$) and test ($\mathcal{D}_{\text{test}}$) sets, which are mutually exclusive. The model is then fit only on $\mathcal{D}_{\text{train}}$ minimizing the cost function, in order to find the best set of parameters $\hat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \{C(\boldsymbol{y}_{\text{train}}, f(\boldsymbol{X}_{\text{train}};\boldsymbol{\theta}))\}$ (we call E_{in} the value of the cost function for the best fit model on the training set). Eventually, performances are evaluated by computing the cost function of the trained model on the test set (E_{out}). This procedure assures that we get an unbiased estimate for the model predictive performances. In fact, just focusing on the training error E_{in} results do not reflect real score in new scenarios. This happens because the algorithm has already seen those data during training, and hence the evaluation tests how much it has memorized the available data, instead of how much it is able to generalize its knowledge. The scenario where performances are good on the training set but highly deteriorate on the test is called overfitting.

Fig. 2.1 shows three different cases of the performances of a learning model:

- **[a]** underfitting occurs when the model is too simple to describe our data: training error is large, and there are no generalization capabilities;
- **[b]** a good model performs well on the training data and is able to predict the behavior outside the training range. In fact, the model predicts values near to the true function even in regions where there are no training points. In this scenario, the gap between training and test errors is small;
- **[c]** overfitting happens when the model complexity is high for the task: our learner just memorizes training patterns and cannot be generalized to unseen data. Therefore, the training error is small, while test one is large.



(a) *The proposed model is too simple to describe the data: the training error E_{in} is big, and even training data are not explained by the algorithm (underfitting).*

(b) *The model describes very well our data and is able to correctly predict the true function even in unseen scenarios. This is the goal when training an algorithm.*

(c) *The model performs very well in training data points, but fail to understand the behaviour in unseen scenarios. Model complexity is too high for the task, and the gap between training and test error is big (overfitting).*

Figure 2.1: *Three different scenarios that can happen when training a model. Orange line represents the true data distribution, which is unknown. Available data are represented by blue points which do not follow exactly the underlying distribution due to the presence of noise. Blue line indicates the prediction of the model. The goal is to have an algorithm able to explain the available data, but also capable to predict the behaviour in unseen scenario. Visually, it is equivalent to have the minimum possible difference between the orange and the blue line on the whole domain and not only around training points.*

The balance between under and overfitting can be controlled by tuning the model complexity, in the so-called *Bias-Variance trade-off*. The error E_{out} on unknown data decreases as a function of the number of training examples since the sampling noise diminishes and the available samples become more representative of the true distribution underlying the data. The value reachable in the limit of infinite training points is called bias. The variance instead represents the error induced during training due to the sampling noise related to the finite amount of the training set. While the bias can be reduced by adopting models with higher complexity, we note that instead the variance will increase leading to worse predictions in unseen data. Fig. 2.2 displays this behavior; E_{out} can be used as an indicator to set the correct model complexity. This variable initially decreases with the model complexity, and after reaching a minimum (that represents the optimum spot) it starts raising since the algorithm begins to memorize training data and loses its ability to generalize its predictions.

2.2 Model Selection and Parameters Tuning

Sec. 2.1 focuses on the importance of evaluating the performance of a model over data not used in the training procedure. For this purpose, the dataset is split into training and test set, and the latter one is used to compare performances between different models. It is common to further split the training

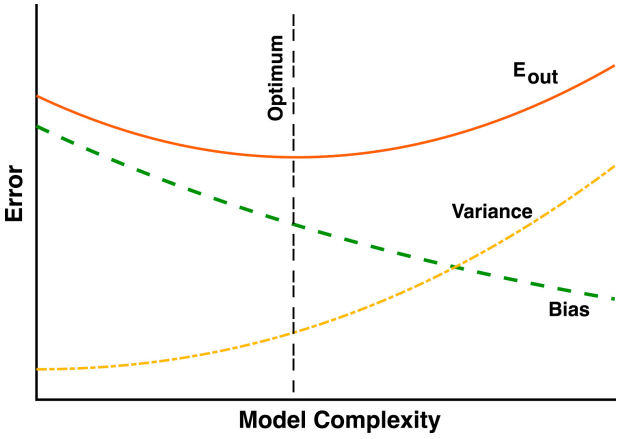


Figure 2.2: *Typical behaviour of the test error E_{out} as a function of the model complexity. The bias decreases with model complexity. while the variance has an opposite behavior. Optimal performance is achieved in intermediate levels of model complexity.*

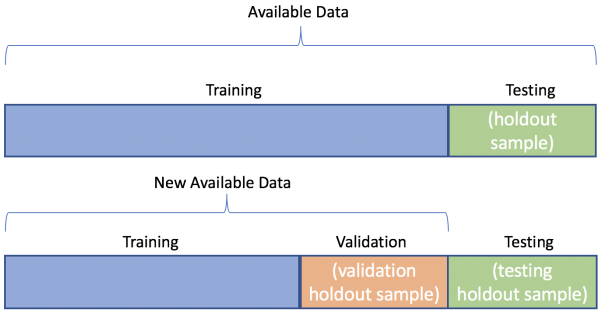


Figure 2.3: *Typical subdivision of the available dataset into training, validation and test set.*

dataset into two subsets: training and validation (see Fig. 2.3). The validation set is commonly exploited as independent dataset to monitor the training (to avoid under and overfitting) or to tune model parameters.

Partitioning the available data however reduce the number of example that the model can see during the training phase; in addition, performances may depend on the random choice of the splits. Cross-validation (CV) is a technique that aims to overcome this problem to make the training more robust. This solution requires splitting the data into training and test. Training one is then split into k subsets (k-fold CV). For each of the k "folds" the model is first trained on the remaining $k-1$ and then validated on the additional part of the data (Fig.2.4). The performance is measured by the average of the values computed for each of the k splits.The best combination of the model parameters is chosen using the CV score; usually, the best algorithm is then retrained on the whole training set, and the final score is calculated on the test set.

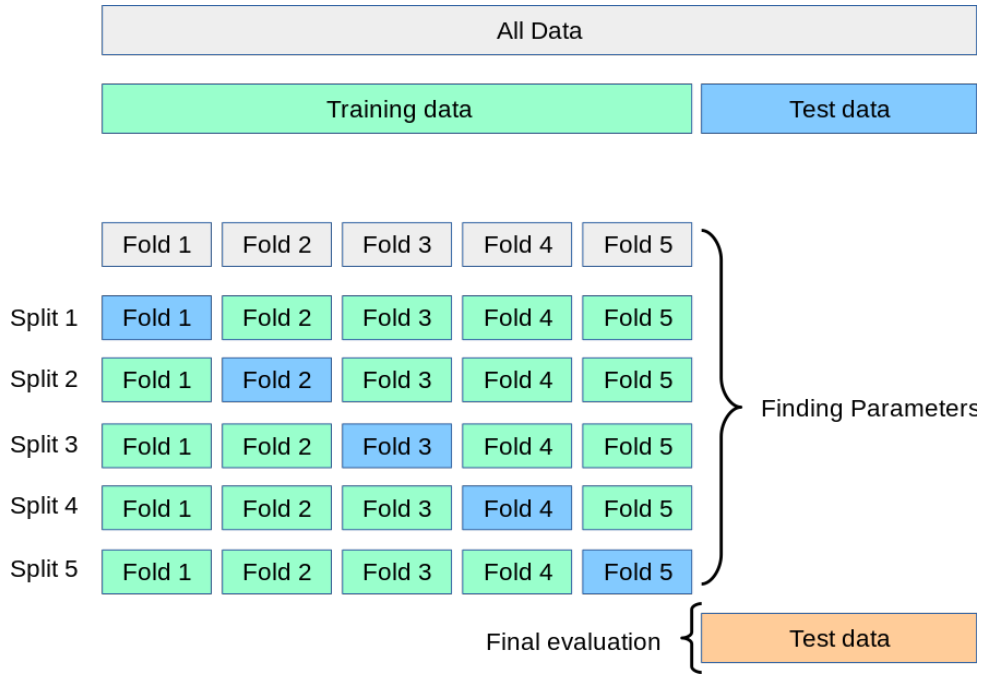
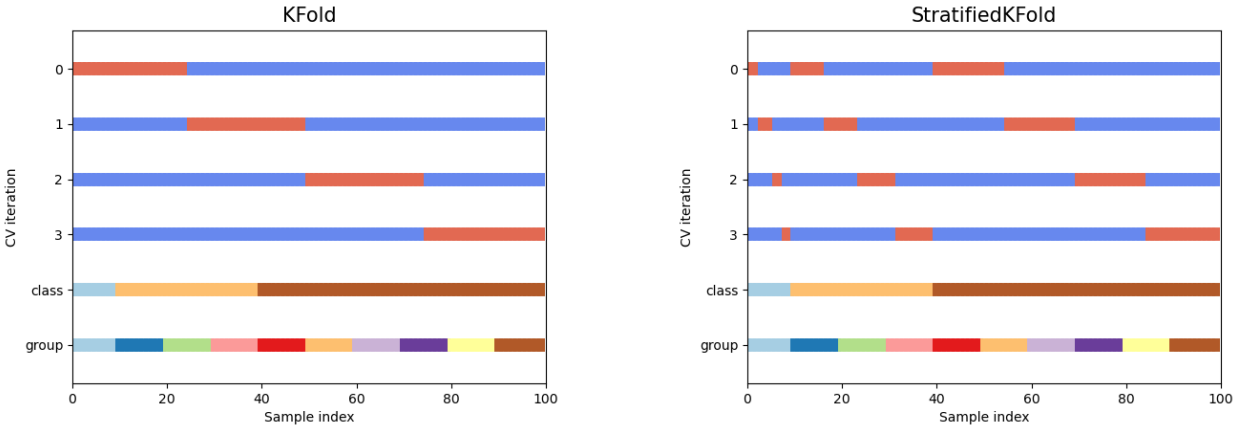


Figure 2.4: *Subdivision of the dataset following the k-fold cross-validation scheme (in this example $k = 5$). Model parameters are tuned using the results of the CV. Best model is then retrained on the whole training set and the final performance is evaluated on the test one.*

In the work described in this thesis, a variation of the CV is used, that is the Stratified k-fold CV. It differs from the standard procedure because it returns stratified folds, which means that each set approximately repeats the same distribution of each target class of the complete set. A visual representation of such difference is given in Fig. 2.5.



(a) *Standard k-fold splits data in different parts without keeping the same percentage of examples from each class as in the entire dataset. For example, in the split 0 the model is trained without seeing any representative of the light-blue class, but it is validated over them.*

(b) *Stratified k-fold maintains the same percentage of each target class in each subset as the complete set.*

Figure 2.5: *Comparison between the splits obtained by standard k-fold and its stratified variant in the case $k = 4$. Data points belong to three different classes (light-blue, orange, brown). For each CV iteration blue line represents data used for training the model, while the red one the examples for evaluating the performances.*

2.2.1 Hyper-parameters tuning

Hyper-parameters are parameters that are not learned during the training procedure, but instead, they control the learning process. Learning rate or the number of neurons inside a Neural Network are examples of these variables; even the model architecture is a hyper-parameter, since it is defined a priori and not discover during training. In order to get the best performances, the parameters space should be searched to get the best score. There are mainly two approaches that can be followed: Grid Search and Randomized Search [5]. These can be applied in conjunction with the CV technique described in Sec. 2.2.

Grid Search systematically explores the parameters space, testing all the combinations of hyper-parameters supplied. This can be useful when there is a limited number of parameters. However, it can be expensive in terms of time and resources, since every combination is tested.

In Randomized Search, hyper-parameters values are sampled at random from a distribution of possible values. This allows to set a budget (i.e. a total amount of rounds) independent of the number of parameters, and it is efficient in the case some hyper-parameters have an higher influence during training and hence result more important to optimize (Fig. 2.6).

2.2.1.1 Bayesian Optimization

The hyperparameter tuning procedure involves finding the best architecture in order to reach the minimum of the cost function. Therefore this problem is related to the field of function optimization. In particular, we would like to perform some sort of random search in which future parameter configurations are sampled from the parts of the search space that are more likely to contain the extrema of the cost function.

The Bayesian optimization approach uses Bayes Theorem to tackle this task when the objective function to optimize is complex, noisy, or expensive to calculate [6]. Bayes Theorem states that the

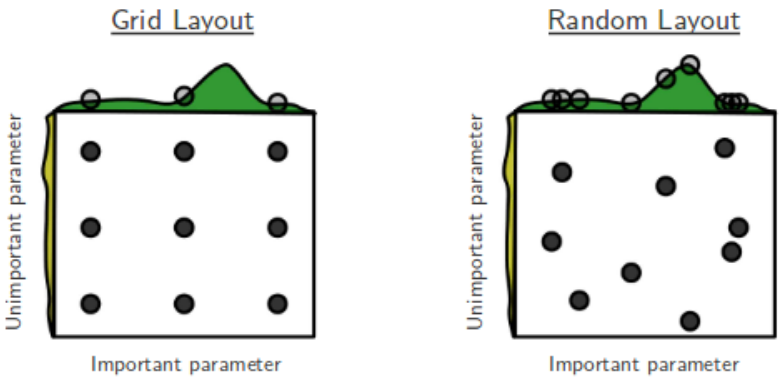


Figure 2.6: *Difference between Grid and Random Search in the case where a parameters is far more important than the other.*

posterior probability of a model M given evidence (observations) E is proportional to the likelihood of E given M multiplied by the prior probability of M:

$$P(M|E) \propto P(E|M)P(M)\,.$$

Let's $\boldsymbol{x}_i$ be the i-th sample from the domain space (that will be a given combination of hyper-parameters), and $f(\boldsymbol{x}_i)$ the value of the objective function at $\boldsymbol{x}_i$. We can define the prior from the collection $\mathcal{D}_{1:t} = \{\boldsymbol{x}_{1:n}, f(\boldsymbol{x}_{1:n})\}$ [1] and then obtain the posterior distribution:

$$P(f|\mathcal{D}_{1:t}) \propto P(\mathcal{D}_{1:t})P(f)\,.$$

The posterior is an approximation of the objective function and can be used to direct future sampling. This belief can be applied through the acquisition function (which refers to techniques to determine how much desirable is to evaluate a point given our present model) to sample from the area of the search space that is more likely to pay off. New samples are then added to $\mathcal{D}$ and the posterior is updated. The process is repeated until the extrema of the objective function is reached or the resources (i.e. a fixed maximum amount of repetitions) are exhausted.

Applying these ideas to parameters tuning we can develop a Random Search algorithm that suggests successive parameter combinations in a clever way, instead of casually spanning the hyper-parameter space. In Python, a possible implementation is done by the package `Ray Tune` [7], which acts as a wrapper of other available packages (in particular `scikit-optimize` [8] is mentioned, as it is the one implemented in this work).

2.2.2 Imbalanced datasets

Considering a classification problem, the number of samples from each class may determine the performance of the algorithm. If classes are not equally represented in the available samples, the dataset is defined imbalanced. In particular, if this disparity is very emphasized, the trained model will end up being biased.

The following metrics are useful to analyze the model results [9]:

- Accuracy is the percentage of examples correctly classified:

$$\text{Accuracy} = \frac{\text{true samples}}{\text{total samples}}$$

  In the case of an unbalanced dataset, this metric should be monitored for all the classes separately, in order to understand whether the performances are good even for the minority groups. It is possible to define a balanced version, which weights each sample according to the inverse prevalence of its class.

[1] Subscripts have the following meaning: $y_{1:n} = \{y_1, \ldots, y_n\}$

- Precision is the percentage of predicted positives that were correctly classified:

$$\text{Precision} = \frac{\text{true positive}}{\text{true positive} + \text{false positive}}$$

  It describes which is the proportion of correct positive identifications.

- Recall is the percentage of actual positives that were correctly classified:

$$\text{Recall} = \frac{\text{true positive}}{\text{true positive} + \text{false negatives}}$$

  It identifies the proportion of actual positives that are classified correctly.

- F1 is the weighted average of the Precision and Recall:

$$\text{F1} = 2\,\frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

- The Receiving Operating Curve (ROC) plots True Positive Rate vs. False Positive Rate at different classification thresholds. The Area Under the Curve (AUC) displays the probability that a classifier will rank a random positive sample higher than a random negative sample.

As an example, let us consider a binary classification problem where 99% of the dataset belongs to class A and the remaining 1% to class B. Just looking at accuracy, a model that only outputs class A has a score of 99%; hence it may be chosen as the best model even though it misclassifies all the examples in the minority class. During the training phase it is of paramount importance to monitor these metrics on the validation set; early stopping the training phase may be helpful to acquire a model with higher generalization capabilities.

In addition, custom loss functions can assist the learning. For example, a good idea is to heavily weigh the examples in the minority class: in this way, the model will pay more attention to them. As suggested in [9], a possible weight for class i can be: $w_i = 0.5(\text{total samples})/(\text{class } i \text{ samples})$.

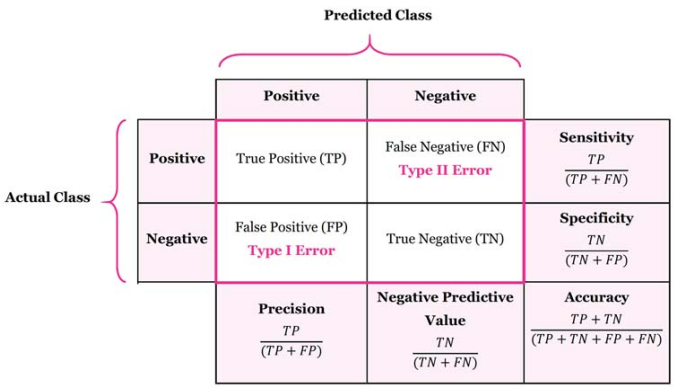
Some visualization means offer a graphical representation of the performances. Confusion matrix (Fig. 2.7a) displays as columns the instances in a predicted class, while rows exhibit the instances of the actual class (or vice-versa). Hence, the element (i, j) is the number of observations actually in group i but predicted in group j. A perfect classier will have a diagonal confusion matrix, where each sample is correctly assigned to the true class.

The Receiver Operating Characteristic (ROC) curve illustrates the ability of a binary classifier to discriminate the two classes as the threshold is varied. In fact, a binary classifier, instead of outputting the row class label, can output a probability of the example to belong to a certain class; then fixing a threshold we can assign it to class A if the probability is under this value or to the other otherwise. ROC is created by plotting true positive rate (TPR) against false positive rate (FPR) at various thresholds. A random classifier shows as a curve the bisector, since with equal probability assigns samples to one class or another, and its AUC score is 0.5. The higher the curve is with respect to the bisector, the better is the classifier, as shown in Fig. 2.7b.

2.2.2.1 Oversampling: SMOTE

One possible way to mitigate the disproportion between classes is to generate new samples from under-represented ones. The most basic strategy involves generating new entries by randomly sampling with replacement from the available samples. This results in all classes being equally represented during the tuning procedure. Other techniques instead generate new samples by interpolating existing data. An example of such algorithms is the Synthetic Minority Oversampling Technique (SMOTE) [10]. Given a sample x_i, its k nearest-neighbors are considered to generate the new sample x_{new}; one of them (x_{zi}) is randomly picked and then the synthetic example is given by:

$$x_{\text{new}} = x_i + \lambda(x_{zi} - x_i)\,, \quad \lambda \in [0, 1] \text{ at random}$$



(a) *Confusion matrix in the case of a binary classification. Rows represents the actual classes, while columns the predicted ones. Entries in the diagonal are the correctly classified samples.*

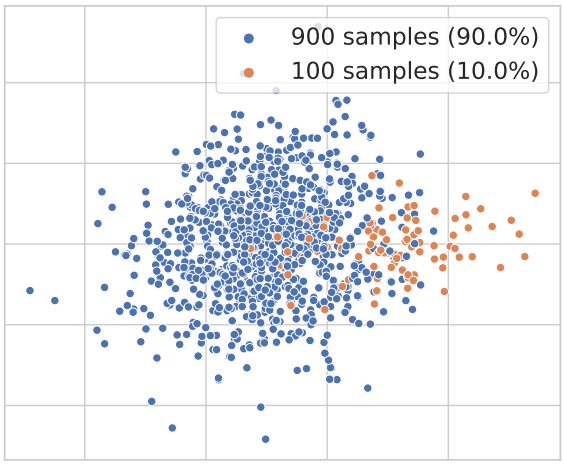
(b) *Example of ROC curves. A random classifier displays the bisector as its curve. The more is the AUC, the better is the model: in the example, blue ones performs better than orange. A perfect classifier has an AUC score equal to 1, since it always predict the correct label.*

Figure 2.7: *Visual representation such as confusion matrix and ROC curve may help to understand the real performances of a model in the case of unbalanced dataset, in addition to the simple accuracy score.*

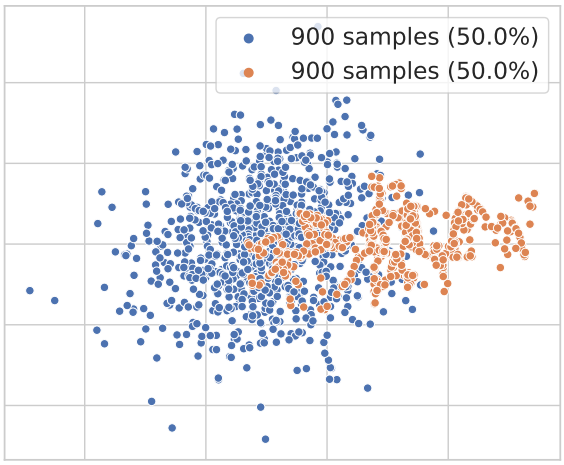
New generated features lie in between the line that connects to real samples; the synthetic examples are relatively close in feature space to existing ones, so they are plausible.

This procedure is applied to all points of the minority class; the number of nearest neighbors chosen depends on the amount of oversampling required. This approach forces the decision regions of the minority class to become more general. Fig. 2.8 shows an example of this oversampling technique.

Python implementation can be found in the `imbalanced-learn` package [11].



(a) *Original imbalanced dataset.*

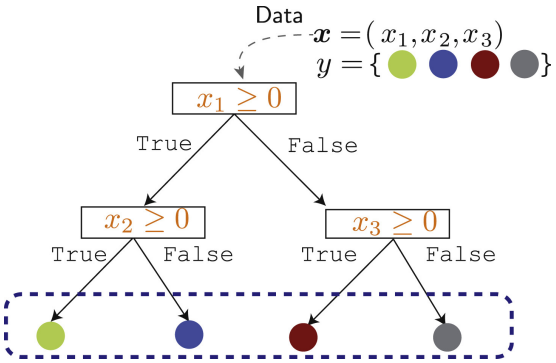


(b) *Oversampling over the minority class with SMOTE.*

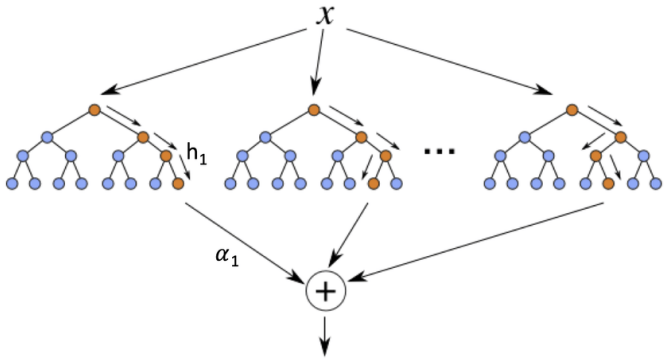
Figure 2.8: *Synthetic data points generated by SMOTE algorithm to equalize an imbalanced daatset.*

2.3 Boost Decision Trees

A Decision Tree (DT) is a non-parametric supervised learning method. It hierarchically partitions the data by learning simple decision rules inferred from data features themselves (Fig. 2.9a). Data are split in the nodes into smaller subsets until reaching the leaves, which correspond to the final partitions.



(a) *DTs consist of root, decision and terminal nodes. In each one data are partitioned until reaching the leaves, which determine the output label.*



(b) *Combining DTs through bagging or boosting creates ensambles classifiers. In bagging final results comes from a majority voting approach, while in boosting predictions of individual trees are summed.*

Figure 2.9: *Examples of a decision tree and an ensable of DTs.*

DTs can easily become complex and end up overfitting; to overcome this common issue with DTs, almost every application implements some regularizations. Stopping criteria help to decide whether to continue dividing the tree or to turn a vertex into a final leaf; tree pruning instead consists of growing an overfitted tree and then delete leaves based on selection criteria.

A single DT has high variance since it is extremely sensitive to lots of details of training data; as a consequence, it is a weak classifier. However it can be easily randomized, and hence it is suitable for ensemble methods, where individual predictions are aggregated to form the final conclusion.

DT ensembles are combinations of DTs through bagging or boosting (Fig. 2.9b). Trees are trained over bootstrapped samples from the training data: the variance is averaged over all the base algorithms, but correlation arises. Random Forest is an example of such a technique, in which also the splitting on each node is found either from all input features or a random subset of them.

In boosting each of the base algorithms is added sequentially, in order to pay more attention to training tuples that were misclassified by previous algorithms. Popular boosting algorithms implement Gradient-Boosted Trees, which use gradient descend methods to optimize the loss function: successive trees are added in order to move in the direction of the gradient. One of the state of the art implementation resides on `XGBoost` (Extreme Gradient Boosting) [12].

Learning the trees consists of building nodes that determine informative partitions at the leaf level about the class labels. In order to achieve that, a parametrization of them along with a cost (or objective) function needs to be defined. Decision tree j with T leaves is parametrized by a function

$$f_j(\boldsymbol{x}) = \omega_{q(\boldsymbol{x})}\,, \quad \boldsymbol{\omega} \in \mathbb{R}^T\,, \quad q : \boldsymbol{x} \in \mathbb{R}^d \to \{1, 2, \ldots, T\}\,, \tag{2.1}$$

where $\boldsymbol{\omega}$ is a weight vector and q maps each data point to the corresponding leaf.
Cost function is generally made up by two terms:

$$\mathcal{C} = \underbrace{\sum_{i=1}^{N} l(y_i, \hat{y}_i)}_{\text{loss}} + \underbrace{\sum_{j=1}^{T} \Omega(f_i)}_{\text{regularization}}\,, \tag{2.2}$$

where i runs over data points and j over decision trees in the ensemble. `XGBoost` implements a regularization term in the form of

$$\Omega(f) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{M} \omega_j^2\,, \tag{2.3}$$

which penalizes both large weights on the leaves and large partitions with many leaves.
In boosting, each tree is added iteratively; the prediction value at step t is

$$\hat{y_i}^{(t)} = \hat{y_i}^{(t-1)} + f_t(\boldsymbol{x}_i)\,. \tag{2.4}$$

and the cost function can be rewritten as

$$\begin{aligned} \mathcal{C}_t &= \sum_{i=1}^{N} l(y_i, \hat{y_i}^{(t-1)} + f_t(\boldsymbol{x}_i)) + \Omega(f_t) \\ &\approx \mathcal{C}_{t-1} + \Delta\mathcal{C}_t\,, \end{aligned} \tag{2.5}$$

with the t-th tree that minimizes $\Delta\mathcal{C}_t$. Since for large t each decision tree can be seen as a small perturbation, we can approximate the cost function using a Taylor expansion at the second order. Hence the optimization goal becomes:

$$\mathcal{C}_t = \sum_{i=1}^{N} \left[g_i f_t(\boldsymbol{x}_i) + \frac{1}{2} h_i f_i^2(\boldsymbol{x}_i) \right] + \Omega(f_t)\,, \qquad \text{with: } \begin{cases} g_i = \partial_{\hat{y_i}^{(t-1)}} l(y_i, \hat{y_i}^{(t-1)}) \\ h_i = \partial^2_{\hat{y_i}^{(t-1)}} l(y_i, \hat{y_i}^{(t-1)}) \end{cases} \tag{2.6}$$

Plugging in Eq. 2.3, it is possible to found the best values of the weights and the best objective reduction:

$$\omega_j^* = -\frac{G_j}{H_j + \lambda}\,, \qquad \mathcal{C}^* = -\frac{1}{2} \sum_{j=1}^{T} \frac{G_j^2}{H_j + \lambda} + \gamma T\,, \tag{2.7}$$

with $G_i = \sum_{i \in I_j} g_i$ and $H_j = \sum_{i \in I_j} h_i$, where I_j is the set of points $\boldsymbol{x}_i$ that is mapped to leaf j.
Eq.2.7 measure the goodness of structure $q(x)$. Theoretically, the tree among every possible tree that minimizes $\mathcal{C}^*$ should be chosen; in practice, an approximate greedy[2] algorithm is implemented, which leads to a model that is a good local minimum of the objective function.

2.4 Neural Networks

Artificial Neural Network (NN) is a computational model inspired by the structure of our brain. A generic NN is made of a number of units called "neurons" linked together in a communication network. This basic component takes a vector of features $\boldsymbol{x}$ as input and produces a scalar output $f(\boldsymbol{x})$ (Fig. 2.10a). Each feature is weighted by a linear transformation and recentered with the neuron-specific bias b; at the end a non-linear activation function σ is applied to produce the output.

$$a_i(\boldsymbol{x}) = \sigma_i\left(\boldsymbol{\omega}_i \cdot \boldsymbol{x} + b_i\right)\right. . \tag{2.8}$$

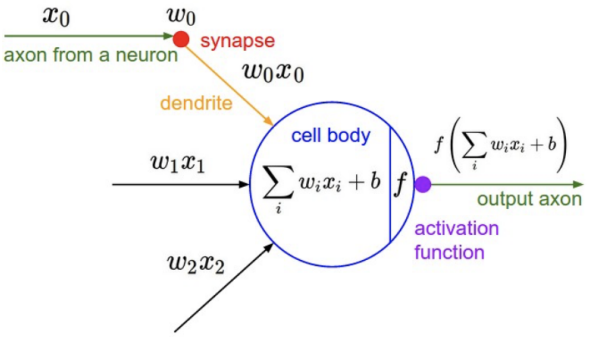
Different kinds of activation functions can be implemented; Fig. 2.10b shows some examples.

Neurons are stacked into layers and the information flows from one layer to another (Fig. 2.10c): the first one is the input, the final is called output layer, while the ones in the middle are the hidden. This is the simplest architecture and defines so-called feed-forward NN, where there are no connections in which outputs of the neurons are fed back into itself. If feedback connections are present, they are called recurrent neural networks.

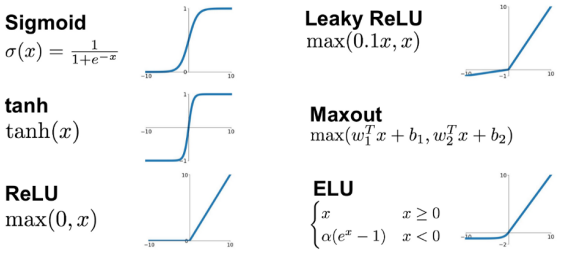
A NN defines a mapping $y = f(\boldsymbol{x}; \boldsymbol{\theta})$ and the learning consists on finding the values of the parameters $\boldsymbol{\theta}$ (weights $\boldsymbol{w}_i$ and biases $\boldsymbol{b}_i$) that best approximate the function [13]. The basic procedure for training defines a loss (cost) function, and it uses gradient descent to minimize it with respect to network parameters (weights and biases). Modern approaches implement learning in the form of maximum likelihood; the cost function is then the negative log-likelihood of the data

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{x,y \sim \hat{p}_{\text{data}}} \log\left[p_{\text{model}}(\boldsymbol{y}|\boldsymbol{x})\right]\,. \tag{2.9}$$

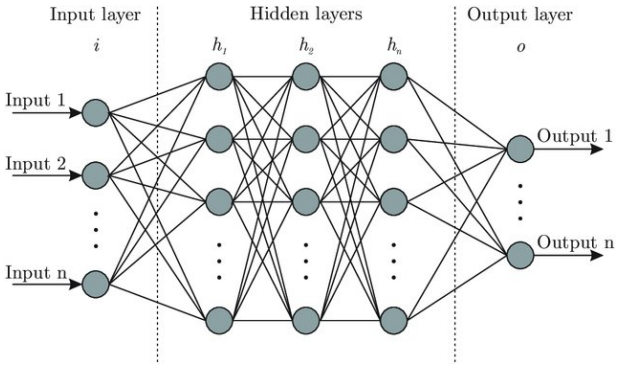
[2]An algorithm is defined as greedy if at each step it takes the best immediate, or local, solution while finding an answer. As consequence, the global optimal solution is approximate by locally optimal choices.



(a) *Schematic representation of an artificial neuron. The inputs are weighted by their relative importance, and then a non-linear transformation is applied (activation function).*



(b) *Different examples of common activation functions implemented in NN.*



(c) *Typical structure of the neurons organization inside a NN; input and output layer are highlighted, along with a number (3 in this specific example) of hidden ones.*

Figure 2.10: *Basic architecture and components of a NN, along with common activation functions.*

Brute force calculation is not suitable for NNs, since it would require to calculate a gradient for each parameter at each iteration of the algorithm. Instead, back propagation algorithms [14] are implemented; they exploit NN structure to efficiently compute the gradient using the derivative chain rule [4]. Denote with $l = 1, \dots, L$ the layer index, ω_{jk}^l the weights from the k-th neuron in layer $l-1$ to the j-th in layer l and b_j^l the bias of the neuron. The activation of the j-th neuron in the l-th layer is:

$$a_j^l = \sigma\left(\sum_k \omega_{jk}^l a_k^{l-1} + b_j^l\right) = \sigma(z_i^l) \tag{2.10}$$

The change of the cost function E of the j-th neuron in the l-th layer is

$$\Delta_j^l = \frac{\partial E}{\partial z_j^l} = \frac{\partial E}{\partial a_j^l}\sigma'(z_j^l) \tag{2.11}$$

$$= \frac{\partial E}{\partial b_j^l}\underbrace{\frac{\partial b_j^l}{\partial z_j^l}}_{1} \tag{2.12}$$

$$= \left(\sum_k \Delta_k^{l+1}\omega_{kj}^{l+1}\right)\sigma'(z_j^l)\,. \tag{2.13}$$

Finally, differentiating the cost function with respect to the weights ω_{jk}^l:

$$\frac{\partial E}{\partial \omega_{jk}^l} = \frac{\partial E}{\partial z_j^l}\frac{\partial z_j^l}{\partial \omega_{jk}^l} = \Delta_j^l a_k^{l-1}\,. \tag{2.14}$$

Summing up, the algorithm is composed by two parts:

1. **Forward step**. Starting from the input, calculate activations a_j^1 in the first layer, propagate through the architecture (Eq. 2.10) and calculate the error in the top layer with Eq. 2.11 with $l = L$.
2. **Backward step**. Backpropagate the error with Eq. 2.13 and calculate the gradient using Eq. 2.12 and 2.14.

This procedure is computationally efficient, but still contains some critical issues, such as the problem of the vanishing gradient, which may be tackled by choosing proper activation functions that are able to mitigate the problem.

2.4.1 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a type of NN specialized in processing sequence of values $\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(\tau)}$ [13]. This can be achieved by the presence of feedback connections, which allow sharing parameters across different parts of the model. This architecture behaves like a dynamical system driven by an external signal $\boldsymbol{x}^{(t)}$, which state is described by the equation

$$\boldsymbol{s}^{(t)} = f(\boldsymbol{s}^{(t-1)}, \boldsymbol{x}^{(t)}, \boldsymbol{\theta})\,. \tag{2.15}$$

The current state clearly preserves information about the whole past sequence. Hidden units state $\boldsymbol{h}$ of an RNN can be described by the Eq. 2.15; to make predictions, extra features are built from the hidden state $\boldsymbol{h}$ and are arranged into output layers. Hence, the state of the RNN and its output at time t are

$$\begin{aligned} \boldsymbol{h}^{(t)} &= f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}, \omega_h) \\ \boldsymbol{o}^{(t)} &= g(\boldsymbol{h}^{(t)}, \omega_o)\,, \end{aligned} \tag{2.16}$$

where f and g represent transformations of hidden and output layers, and ω_h and ω_o their weights. The recursive structure of a RNN can be unrolled over time in order to transform the architecture into a direct acyclic graph, as shown in Fig. 2.11; thus, a temporal network can be interpreted as

deep neural network with parameters sharing. Each timestep of the unrolled graph may be seen as an additional layer given the order dependence of the problem and the internal state from the previous timestep is taken as an input on the subsequent timestep. This description allows applying

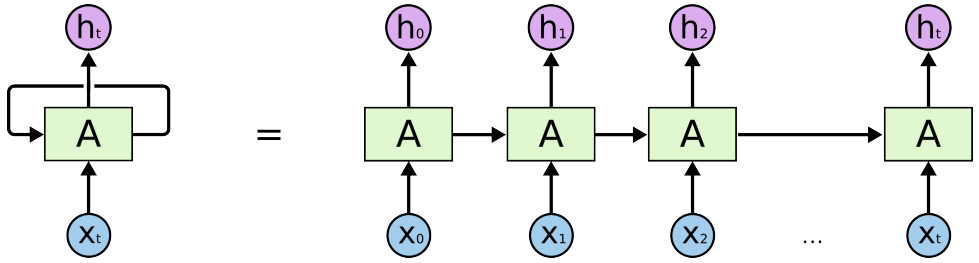


Figure 2.11: *RNN process the input $\boldsymbol{x}$ incorporating information in its hidden state and passes it through time. The computation graph can be unfolded, in order to associate each node with a single time instance.*

the notions of backpropagation to the unrolled graph, creating the so-called backpropagation through time (BPTT) algorithm. The forward step involves looping through the triplets $(\boldsymbol{x}^{(t)}, \boldsymbol{h}^{(t)}, \boldsymbol{o}^{(t)})$ defined in Eq. 2.16 [15]; the discrepancy between the output and the true label is evaluated by the objective function among all T timesteps:

$$L(\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(T)}, \boldsymbol{y}^{(1)}, \dots, \boldsymbol{y}^{(T)}, \omega_h, \omega_o) = \frac{1}{T} \sum_{t=1}^{T} l(\boldsymbol{y}^{(t)}, \boldsymbol{o}^{(t)}) \,. \tag{2.17}$$

The backpropagation passage requires

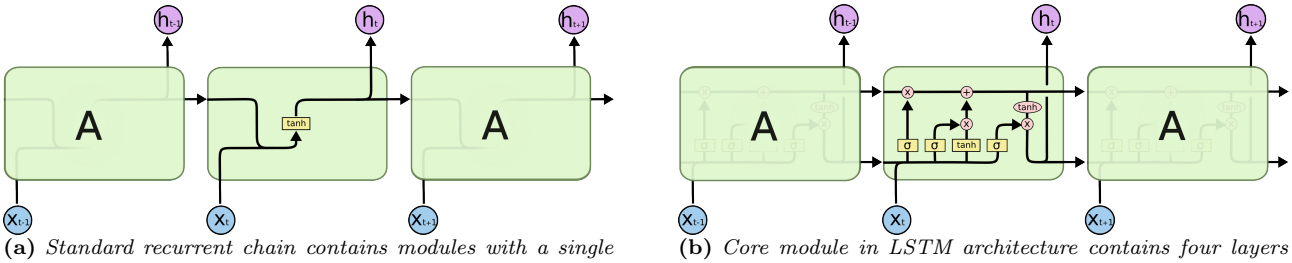
$$\begin{aligned} \frac{\partial L}{\partial \omega_h} &= \frac{1}{T} \sum_{t=1}^{T} l \frac{\partial l(\boldsymbol{y}^{(t)}, \boldsymbol{o}^{(t)})}{\partial \omega_h} \\ &= \frac{1}{T} \sum_{t=1}^{T} \frac{\partial l(\boldsymbol{y}^{(t)}, \boldsymbol{o}^{(t)})}{\partial \boldsymbol{o}^{(t)}} \frac{\partial g(\boldsymbol{h}^{(t)}, \omega_h)}{\partial \boldsymbol{h}^{(t)}} \frac{\partial \boldsymbol{h}^{(t)}}{\partial \omega_h} \,, \end{aligned} \tag{2.18}$$

where the last term $\partial \boldsymbol{h}^{(t)} / \partial \omega_h$ can be derived by the chain rule. The full computation is demanding in term of resources, and easily leads to vanishing or exploding gradients; in practice it is common to use strategies that truncate the calculation into shorter pieces, considering only τ steps (truncated BPTT).

2.4.1.1 Long-Short Term Memory networks

RNNs are able to learn short-term temporal dependencies, but they fail when the context requires recalling information with a large time gap. This could be related to gradient problems (vanishing or exploding) or to the exponentially smaller weights given to long-term interactions with respect to closer ones.

Long-Short Term Memory networks (LSTMs) are a special kind of RNNs specifically designed to handle long term information and to avoid the issue concerning the vanishing gradient. This is possible due to the implementation of gated units, which learn how much information needs to be preserved in temporary memory and how much it should be propagated in time. Fig. 2.12 [16] shows the difference between the core module of a standard RNN compared to an LSTM. The cell state is described by the horizontal line at the top of the unit (Fig. 2.12b) and propagates information over time: gates can add or remove details to alter the stream. In particular, these steps summarize the flow inside an LSTM unit:



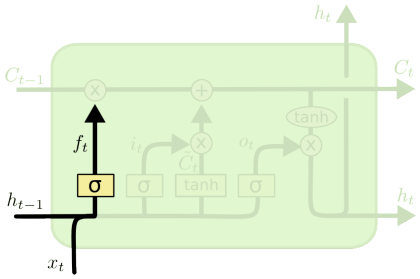
(a) *Standard recurrent chain contains modules with a single layer.*

(b) *Core module in LSTM architecture contains four layers that allow to maintain information into network memory.*

Figure 2.12: *Comparison between the modules in standard recurrent networks and LSTM.*

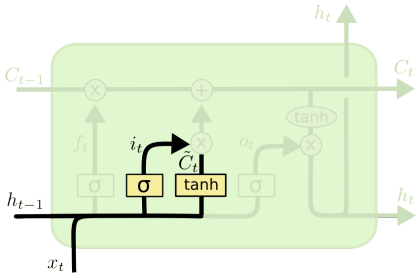
1. **Forget gate** This sigmoid layer decides what information is going to be lost from the cell state. The output of this unit is

$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right) \tag{2.19}$$



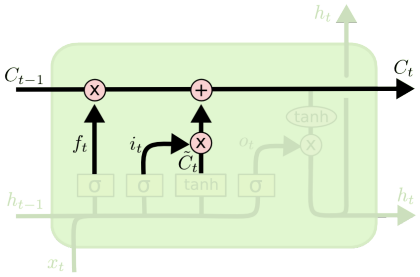
2. **Input gate** Input layer activates with a sigmoid and decides which information i_t is going to be updated. This is merged with the results of a tanh layer, which creates the new vector of candidates $\tilde{C}_t$, and their combination makes the update of the state. The process is drivenby the equations:

$$\begin{aligned} i_t &= \sigma\left(W_i \cdot [h_{t-1}, x_t] + b_i\right) \\ \tilde{C}_t &= \tanh\left(W_C \cdot [h_{t-1}, x_t] + b_C\right) \end{aligned} \tag{2.20}$$



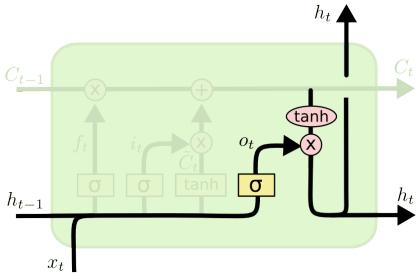
3. **Cell update** The cell state is updated multiplying the old state C_{t-1} by the forget factor f_t and then adding the new information:

$$C_t = f_t C_{t-1} + i_t \tilde{C}_t \tag{2.21}$$



4. **Output gate** The output is a filtered version of the cell state:

$$\begin{aligned} o_t &= \sigma\left(W_o \cdot [h_{t-1}, x_t] + b_0\right) \\ h_t &= o_t \tanh(C_t) \end{aligned} \tag{2.22}$$



2.5 Natural Language Processing

Natural Language Processing (NLP) is a branch of artificial intelligence that focuses on developing machines able to read, understand and manipulate human language. Typical NLP tasks involve for example speech recognition, sentiment analysis, topic extraction and classification, natural language generation.

NLP is a challenging problem since it deals with unstructured data difficult to understand by a machine

since the rules underlying human language imply high levels of abstraction.

This section describes some algorithms implemented in this work to perform both unsupervised topic extraction and supervised text classification. In addition, state-of-the-art solutions are presented, highlighting the so-called "attention" mechanism, which is the foundation of the Transformer networks.

Lastly, the spectrum kernel is reported as a different method to extract features from the text, which differs from the mapping applied by word embedding algorithms.

2.5.1 Latent Dirichlet Allocation

Latent Dirichlet allocation (LDA) is a generative probabilistic model of a corpus. The basic idea is that documents are represented as random mixtures over latent topics, where each topic is characterized by a distribution over words [17]. The corpus is a collection of D documents, each one is a sequence of N words, and K different topics are present. The model is based on the following generative assumptions of the corpus [18]:

1. For each topic $kinK$, the probability of a word to appear in such topic is given by a Dirichlet distribution[3]: $\beta_k \sim \text{Dirichlet}(\eta)$.
2. For each document $d \in D$ draw the topic proportions $\theta_d \sim \text{Dirichlet}(\alpha)$.
3. For each word i in document d:
   (a) The topic is assigned sampling from a multinomial distribution[4]: $z_{di} \sim \text{Multinomial}(\theta_d)$
   (b) The observed word is drawn as $w_{ij} \sim \text{Multinomial}(\beta_{z_{di}})$.

To estimate the parameters, the posterior distribution

$$p(z, \theta, \beta|w, \alpha, \eta) = \frac{p(z, \theta, \beta|\alpha, \eta)}{p(w|\alpha, \eta)} \tag{2.23}$$

of the hidden variables given a document needs to be computed. Since it is intractable for exact inference, approximations are needed. A possible approach is to use variational Bayesian methods to estimate it with a simpler distribution $q(z, \theta, \beta|\lambda, \phi, \gamma)$. The optimizing values of the variational parameters are found by minimizing the Kullback Leibler (KL) divergence[5] between the variational distribution and the true posterior.

In practice, LDA algorithms transforms the document-word matrix into a document-topic one, where for each document there is the probability of belonging to the given topic. LDA requires to specify in advance the number of topics k. Since in the unsupervised setting this is also our goal, the solution is to try different models with distinct values of k, and then compare them using the perplexity, that is a statistical measures of how well a probability distribution predicts a sample. Perplexity is calculated in a holdout set D_{test}, and it is defined as [17]:

$$\text{Perplexity}(D_{\text{test}}) = \exp\left\{-\frac{\sum_{d=1}^{M} \log p(\boldsymbol{w}_d)}{\sum_{d=1}^{M} N_d}\right\} = \exp\left\{-\log \text{likelihood per word}\right\}, \tag{2.24}$$

where M is the number of documents, $\boldsymbol{w}_d$ represents the words in document d, and N_d is the number of words in the document. A lower score indicates better generalization performance, hence a better model.

[3]Dirichlet distribution of order $K \geq 2$ with parameters $\alpha_1, \ldots, \alpha_K > 0$ has a probability density function given by $f(x_1, \ldots, x_K; \alpha_1, \ldots, \alpha_K) = \frac{1}{B(\alpha_1, \ldots, \alpha_K)} \prod_{i=1}^{K} x_i^{\alpha_i - 1}$, where $\sum_{i=1}^{K} x_i = 1$ and $x_i >= 0 \, \forall i \in \{1, \ldots, K\}$ and $B(\alpha_1, \ldots, \alpha_K)$ is the multivariate beta function.

[4]Multinomial distribution is a generalization of a Binomial distribution in the case of k different outcomes. x_j denotes the number of times outcome j occurs in n different trials, and p_j the probability of this outcome. Assuming $\sum_{i=1}^{k} x_i = n$, then the probability mass function is $f(x_1, \ldots, x_k; n, p_1, \ldots, p_k) = \Pr(X_1 = x_1, \ldots, X_k = x_k) = \frac{n!}{x_1! \cdots x_k!} p_1^{x_1} \cdots p_k^{x_k}$.

[5]Kullback Lieber divergence is a measure of the difference between two probabilities ditributions. If P and Q are discrete probability distributions defined in the same probability space $\mathcal{X}$, then the KL divergence is $D_{\text{KL}}(P \parallel Q) = \sum_{x \in \mathcal{X}} P(x) \log\left(\frac{P(x)}{Q(x)}\right)$ [19].

Eventually, clustering documents that share similar topics can be achieved in two ways:

1. Assign each document to the topic with the highest probability score found by LDA.
2. Applying a clustering algorithm such as k-means (see Sec. 2.6) to the document-topic probability matrix.

2.5.2 Word2Vec and Doc2Vec models

Word embedding consists of a representation of a vocabulary of words. It aims to capture the context and the meaning of them, in contrast with other methods such as count vectorizers and term-frequency inverse document-frequency (TF-IDF), which encoding creates features only based on occurrences, discarding information based on relative position of words in sentences.

Word2Vec [20] is a technique to learn word embedding based on a shallow neural network. Words are encoded into vectors such that words with common context are closed in the vector space. Semantic similarity is measured by the cosine similarity[6] between vectors. As displayed in the examples in Fig. 2.13, semantic and syntactic patterns are preserved in the mathematical representation of words by applying algebraic operations.

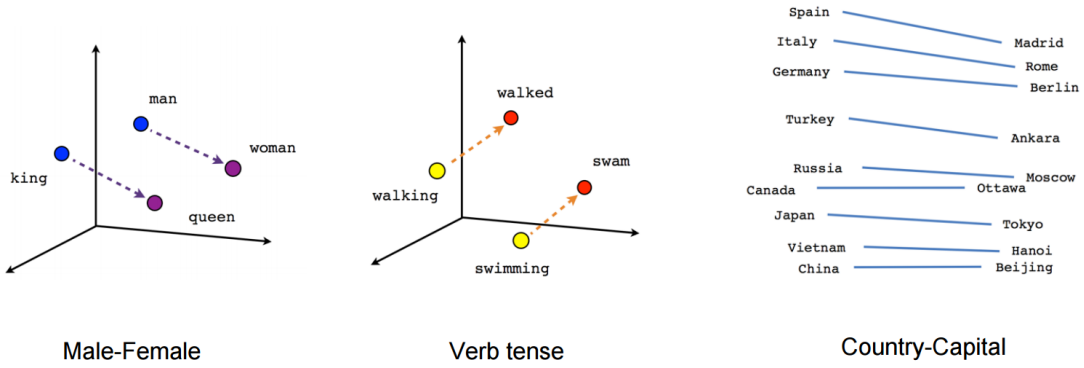


Figure 2.13: *Word embedding generated by Word2Vec is able to reproduce semantic and syntactic relations between words.*

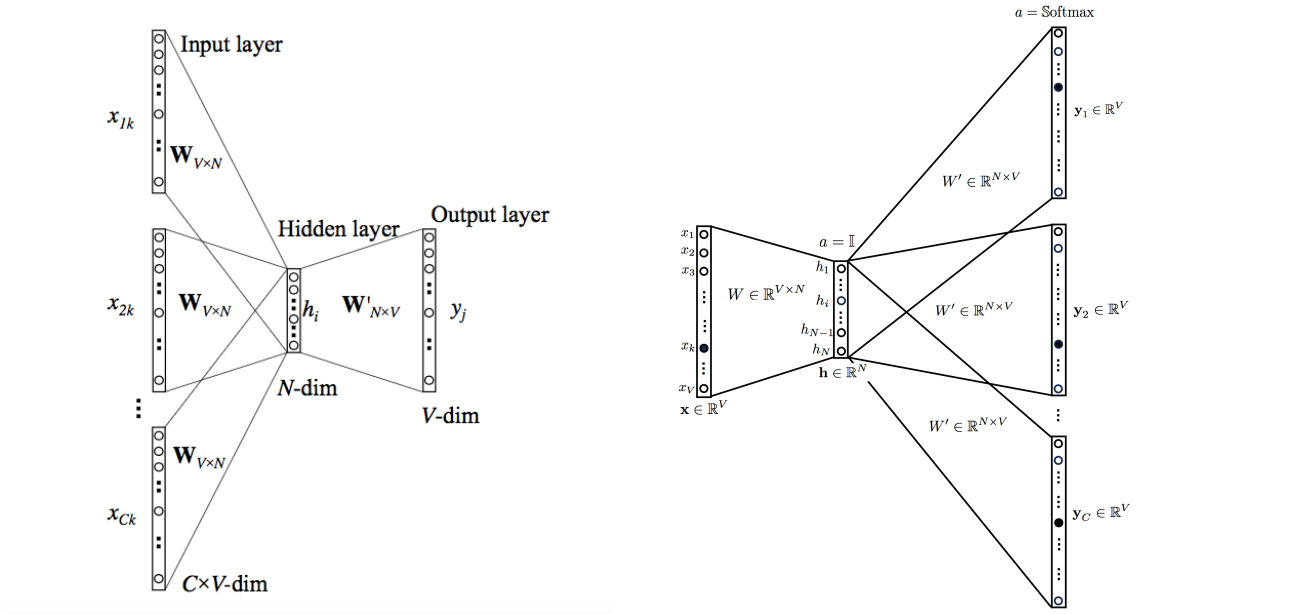
Word2Vec is created using Continuous Bag-of-Words (CBOW) and Skip-Grammar methods. COBW accepts as input the context of a word and tries to predict it; Skip-grammar instead predicts surrounding context words starting from a given one. Fig 2.14 gives a schematic idea of the flow of the two networks. Word2Vec is not interested in the outputs of the architectures, but instead makes use of the hidden state layer to represents words.

Doc2Vec aims to extend the notions of Word2Vec in order to give a numerical representation of a document. The model scheme is similar to COBW, but it adds another feature vector that is unique to each document. During the training of word vectors, the document vector is also trained and it will contain an embedding for the document. This model is the Distributed Memory version of Paragraph Vector (PV-DM) [21]. Another version more similar to Skip-Grammar (Distributed Bag of Words version of Paragraph Vector, PV-DBOW) may be used for the same scope, but PV-DM usually reaches better performances.

2.5.3 State of the art models: BERT

The current state-of-the-art of language model for NLP is BERT (Bidirectional Encoder Representations from Transformers) [22]. It differs from other models in the fact that it applies the concept of bidirectional training (typical of Transformers, see Sec. 2.5.3.1) to language modeling. Previous

[6] The cosine similarity between two vectors $\boldsymbol{A}$ and $\boldsymbol{B}$ is defined as the cosine of the angle θ between them: $\text{sim} = \cos\theta = \frac{\boldsymbol{A}\cdot\boldsymbol{B}}{\|\boldsymbol{A}\|\|\boldsymbol{B}\|}$.



(a) *COBW takes multiple context words to predict the target.*

(b) *Skip-Grammar outputs probability distribution for context words given the input.*

Figure 2.14: *Word2Vec is based on two shallow networks, Continuous Bag-of-Words (CBOW) and Skip-Grammar.*

architectures scanned a text sequence from right to left, left to right, or a combination of the two, but the bidirectional approach seems to lead to a deeper sense of language context.

BERT implements just the encoder part of a Transformer, and these features are then used to make predictions. As stated before, the revolution resides on the bidirectional training idea, which allows learning the context of a word based on all its surroundings. During training two strategies are incorporated to define the prediction goal:

Masked ML (MLM) 15% of the input tokens at random are masked before being fed to the network. The model seeks to predict only the masked words based on the context given by the other ones. In addition, some other strategies are applied. This is due to the fact that the embedded space learned by BERT will be fine-tuned for a specific task; during this second phase though, there are no masked tokens. Hence these strategies try to mitigate this mismatching.

Next Sentence Prediction (NSP) Relationship between two sentences is of fundamental importance in many NLP tasks. During the training process, BERT is fed with pairs of sentences and learns whether the second one is the subsequent phrase in the document. 50% percent of the samples used in training is composed of subsequent sentences, while the other half of the dataset is made up of random ones.

During training both MLM and NSP are trained together, and the minimization of a combined loss is set as the goal. The abstract representation learned in this way can then be fine-tuned to be applied to a wide variety of language tasks.

There exist some variations of BERT model:

RoBERTA (Robustly optimized BERT approach) [23] is a retraining with an improved methodology and more data and computer power. In particular, the NSP step is removed from training, while dynamic masking is introduced (so that masked tokens change during training epochs).

DistillBERT [24] makes use of the knowledge distillation (or teacher-student learning), where a small model is trained to reproduce the behavior of a larger one. This is possible since the distilled model goal is to reproduce the output distribution of the full model.

ELECTRA (Efficiently Learn an Encoder that Classifies Token Replacements Accurately) [25] uses

both a generative (as BERT) and discriminatory model during training. In particular, a small BERT constructs corrupted sequences from the text by guessing the masked token; the discriminator (ELECTRA) predicts whether each token is original or replaced by the generator. In this way learning exploits the whole sentence instead of only the masked words.

2.5.3.1 Attention mechanism: transformers

Attention mechanism is an ML technique that mimics cognitive attention. An attention function maps a query and a set of key-value pairs to an output. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key [26]. Qualitatively it allows the model to focus on the relevant part of the input sequences by constituting an additional context layer.

The Transformer [26] is a model that relies entirely on the attention mechanism instead of recurrence. Its architecture is shown in Fig. 2.15 and features an encoding and decoding stacks.

The implemented attention function is the "Scaled Dot-Product Attention"; the matrix of outputs is:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V\,, \quad (2.25)$$

where Q, K, V are the matrices of queries, keys and values and d_k is the dimension of queries and keys (values have dimension d_v). The actual implementation linearly project the queries, keys and values h times with different, learned linear projections to d_k, d_k and d_v dimensions, respectively. Attention is then performed in parallel, giving d_v-dimensional output values, which are concatenated and projected once again. This allows to capture information from different representation subspaces at different positions.

In the transformer, the attention between encoder and decoder allows every position in the decoder to attend over all positions in the input sequence. The self-attention in the encoder (decoder) instead allows each position in the stack to attend to all positions in the previous layer of the encoder (decoder).

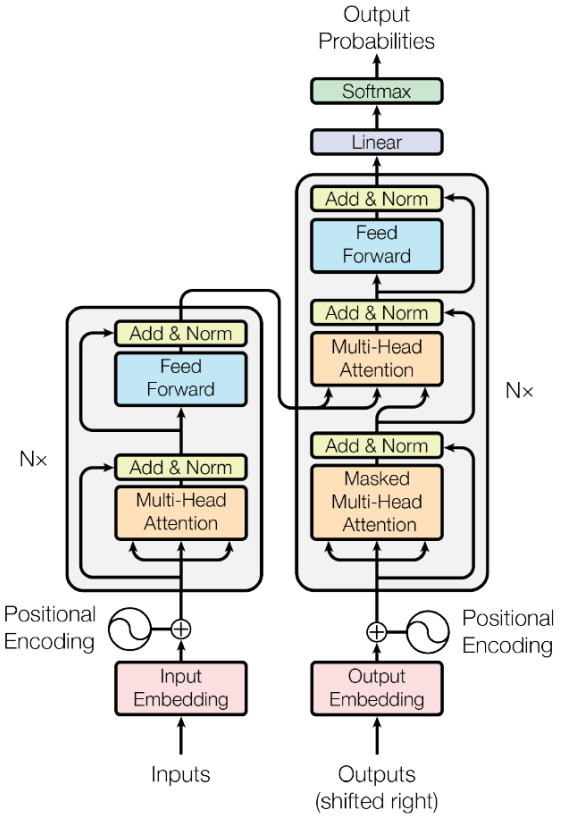


Figure 2.15: *Transformer model architecture (see [26]).*

2.5.4 Spectrum kernels for feature extraction

State-of-the-art NN approaches to NLP focus on word-level embedding. Nevertheless, character-level features can be extracted; they can provide a representation of a word that depends only on its inner structure, that is the sequence of characters. In this work, in particular, spectrum kernel is implemented.

The function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is a kernel if there exists a mapping function $\phi : \mathcal{X} \to \mathbb{K}$ from the input space $\mathcal{X}$ to the kernel space $\mathcal{K}$ such that k can be written as the inner product $k(\boldsymbol{x}, \boldsymbol{z}) = \langle \phi(\boldsymbol{x}), \phi(\boldsymbol{z}) \rangle$.

Following the description in [27], define Σ as as the set of all possible symbols and $\Sigma^* = \bigcup_p \Sigma^p$ the set of strings of arbitrary length. Then, the spectrum kernel $\phi^p : \Sigma^* \to \mathbb{Z}^{|\Sigma^p|_+}$ counts any possible contiguous sub-sequence of length p. The u-th feature computed on an input string $\boldsymbol{x}$ is defined as $\phi^p_u(\boldsymbol{x}) = |(v_1, v_2) : \boldsymbol{x} = v_1 u v_2|$, $u \in \Sigma^p$, where $v_1 u v_2$ denotes the concatenation of strings v_1 , u, and v_2. The p-spectrum kernel is defined as the dot-product between p-spectrum embeddings, that

is $k^p(\boldsymbol{x}, \boldsymbol{z}) = \langle \phi^p(\boldsymbol{x}), \phi^p(\boldsymbol{z}) \rangle$. Hence, spectrum features represents the number of occurrences of a fixed length sub-string. In this way the focus is set to local information rather than on long term dependencies.

Kernel methods have a complexity of the kernel evaluation during training of $O(n^2)$ and the complexity of predictions scales as $O(n)$, since they rely on the Gram matrix:

$$G = \begin{pmatrix} k(x_1, x_1) & k(x_1, x_2) & \cdots & k(x_1, x_n) \\ \vdots & \dots & \ddots & \vdots \\ k(x_n, x_1) & k(x_n, x_2) & \cdots & k(x_n, x_n) \end{pmatrix} \in \mathbb{R}^{n \times n}.$$

To overcome this problem, one possible solution is to approximate the Gram matrix with the Nystrom approximation. It relies on the spectral decomposition $G = U \Gamma U^T$, where $U = [u_1, \dots, u_n]^T \in \mathbb{R}^{n \times n}$ is a set of eigenvectors and $\Lambda = \text{diag}(\lambda_1, \dots, \lambda_n)$ is the eigenvalues matrix. The best rank-k approximation G is given by $G_k = U_k \Lambda_k U_k^T$ with $U_k \in \mathbb{R}^{n \times k}$ and $\Lambda_k = \text{diag}(\lambda_1, \dots, \lambda_k)$, where only the k largest eigenvalues are kept. Calculating G_k though does not ease the computation, since $O(n^2)$ time is required to compute G and $O(n^2 \div n^3)$ for the k-best rank approximation. The Nystrom approximation approximates G_k in $O(n)$ time with $\hat{G}_k = CW^+C^T$, with $C \in \mathbb{R}^{n \times c}$, $W \in \mathbb{R}^{c \times c}$ and W^+ its Moore-Penrose inverse[7].

2.6 K-means clustering

Clustering is an example of unsupervised learning which aims to group unlabelled data into clusters accordingly to some similarity or distance measure [4]. Let $X = \{\boldsymbol{x}_1, \dots, \boldsymbol{x}_N\}$ be a set of N samples, with $\boldsymbol{x}_n \in \mathbb{R}^p$ (p is the number of features). In addition, consider K clusters $\{C_1, \dots, C_K\}$, each one described by its centroid $\boldsymbol{\mu}_k = \text{argmin}_{\boldsymbol{\mu}} \sum_{\boldsymbol{x} \in C_k} d(\boldsymbol{x}, \boldsymbol{\mu})^2 \in \mathbb{R}^p$ ($d(\cdot)$ indicates the distance measure). K-means minimizes the objective function:

$$\mathcal{C} = \sum_{k=1}^{K} \sum_{\boldsymbol{x} \in C_k} d(\boldsymbol{x}, \boldsymbol{\mu}_k)^2 . \tag{2.26}$$

Implementation is done through Lloyd's algorithm:

1. Choose initial centroids.
2. Assign each point to the closest centroid.
3. Create new centroids taking the mean values of all samples assigned to each centroid.
4. Repeat steps 2 and 3 until the difference between old and new centroids is less than a threshold (in other words until there is no significant improvement).

The initialization of centroids is a critical point of the algorithm since it influences both the speed of convergence and whether the result is a local or global minimum. To avoid problems, modern implementations follow `k-means++` variation, which suggests the following procedure to draw initial points [28]:

1a. Take one center c_1 uniformly at random from X.

1b. Select a new center c_k, choosing $x \in X$ with probability $\frac{D(x)^2}{\sum_{x \in X} D(x)^2}$, where $D(x)$ is the shortest distance between x and its closest center.

1c. Repeat the previous step until K centers are drawn.

[7]The Moore-Penrose inverse is a generalization of the inverse matrix. For a matrix $A \in \mathbb{K}^{m \times n}$, $A^+ \in \mathbb{K}^{n \times m}$ is its Moore-Penrose inverse if it satisfies these properties: (i) $AA^+A = A$. (ii) $A^+AA^+ = A^+$. (iii) $(AA^+)^* = AA^+$ (AA^+ is hermitian). (iv) $(A^+A)^* = A^+A$ (A^+A is hermitian).

2.7 Dimensional reduction

Dimensional reduction refers to projecting data that live in an high dimensional space into a lower-dimensional space (called "latent space"), in such a way that some meaningful properties of the original data are retained. In particular, data visualization is infeasible for a large number of features. In this section, Principal Component Analysis and t-Stochastic Neighbour Embedding are described.

2.7.1 Principal Component Analysis

Principal Component Analysis (PCA) decomposes the data in a set of successive orthogonal components in order to find high variance directions [4]. In fact, the relevant information is often described by the directions with the highest variance, while the other components are related to noise and may be ignored since their contribution can be negligible (Fig. 2.16). Denoting with $\boldsymbol{X}$ the data matrix, where the N rows represent the points and the p columns the features, the covariance matrix is

$$\boldsymbol{\Sigma}(\boldsymbol{X}) = \frac{1}{N-1}\boldsymbol{X}^T\boldsymbol{X} \tag{2.27}$$

$$= \boldsymbol{V}\boldsymbol{\Lambda}\boldsymbol{V}^T\,. \tag{2.28}$$

To get Eq. 2.28, $\boldsymbol{X}$ is rewritten using Singular Value Decomposition (SVD)[8] as $\boldsymbol{X} = \boldsymbol{U}\boldsymbol{S}\boldsymbol{V}^T$; then $\boldsymbol{\Lambda} = \mathrm{diag}(\lambda_1, \ldots, \lambda_N)$ with eigenvalues λ_i in decreasing order. The right singular vectors of $\boldsymbol{X}$ are the principal directions of $\boldsymbol{\Sigma}(\boldsymbol{X})$. To reduce the dimensionality to $\tilde{p} < p$, the projection matrix $\tilde{\boldsymbol{V}}_{\tilde{p}}$ is constructed using the singular components described by the $\tilde{p}$ greater singular values; then the projection of the data is $\tilde{\boldsymbol{Y}} = \boldsymbol{X}\tilde{\boldsymbol{V}}_{\tilde{p}}$.

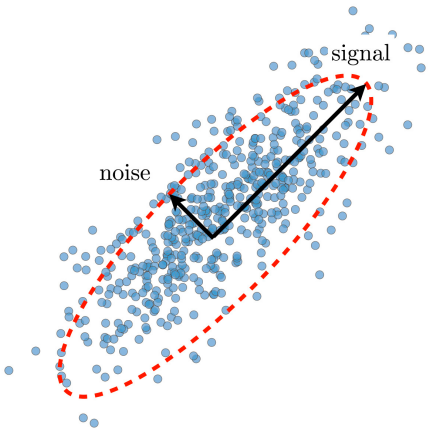


Figure 2.16: *PCA directions with two components. The one with the largest variance is associated to the signal, while the low variance one corresponds to noise.*

2.7.2 t-distributed Stochastic Neighbor Embedding

Linear techniques like PCA do not perform well in preserving local structures of datasets with a large number of clusters; non-linear approaches try to address this issue, as shown in the example in Fig. 2.17.

t-distributed Stochastic Neighbor Embedding (t-SNE) is a non-parametric method that constructs non-linear embeddings where each data point is mapped into a lower-dimensional space in such a way as to preserve local structures of the data [4]. It can very useful for example to discover whether data lie in different clusters.

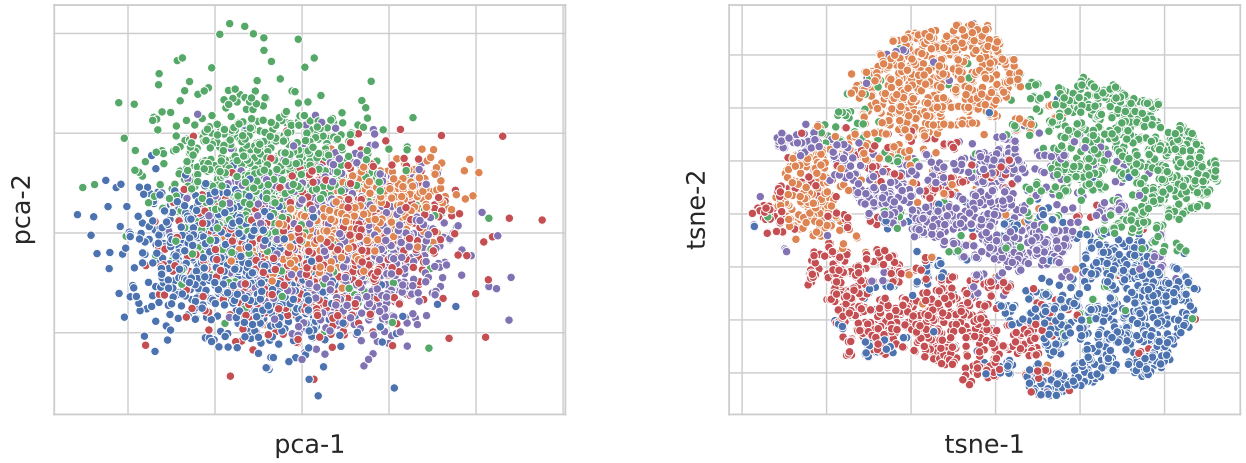
A probability distribution is associated to the neighborhood of each point x (which lives in the space $\mathbb{R}^p$, with p number of features). The likelihood that x_j is the neighbour of x_i is:

$$p_{ij} = \frac{\exp\left(-\frac{\|x_i - x_j\|^2}{2\sigma_i^2}\right)}{\sum_{k\neq i}\exp\left(-\frac{\|x_i - x_k\|^2}{2\sigma_i^2}\right)}\,. \tag{2.29}$$

σ_i are determined by fixing the the perplexity $\Sigma = 2^{H(p_i)}$ (that is the local entropy) equal to all points. In this way points in more dense regions have smaller σ_i. The algorithm then builds a similar distribution probability in a lower-dimensional latent space $\mathbb{R}^{\tilde{p}}$

$$q_{ij} = \frac{(1+\parallel y_i - y_j \parallel^2)^{-1}}{\sum_{k\neq i}(1+\parallel y_i - y_k \parallel^2)^{-1}}\,, \tag{2.30}$$

[8] Singular Value Decomposition factorizes a matrix $\boldsymbol{X}$ as $\boldsymbol{X} = \boldsymbol{U}\boldsymbol{S}\boldsymbol{V}^T$, where $\boldsymbol{S}$ is a diagonal matrix of singular values σ_i, $\boldsymbol{U}$ and $\boldsymbol{V}$ are orthogonal and their columns are respectively the left and right singular vectors of $\boldsymbol{X}$.



(a) *Two principal components of the dataset.*

(b) *t-SNE embedding in a 2-dimensional space.*

Figure 2.17: *Visualization of a synthetic dataset generated with* `sklearn make_classification` *function. It is composed of 5000 samples with 10 features belonging to 5 different classes. (a) shows the two principal components. (b) displays the results of a t-SNE with a 2-dimensional embedded space. t-SNE makes a better job to preserve the local structures of data: in fact, clusters are much more visible than in the PCA result.*

which is chosen to be a long tail one. Consequently short distance information are preserved, while long range interactions are repelled. The coordinates y_i of the low dimensional latent space are found minimizing the Kullback-Liebler divergence between q_{ij} and p_{ij}:

$$D_{\mathrm{KL}}(p \parallel q) = \sum_{ij} p_{ij} \log\left(\frac{p_{ij}}{q_{ij}}\right) . \tag{2.31}$$

3 Failure Prediction

The goal of this study is to build a model able to predict faults before breakdowns happen. The intention is to construct a system that uses Machine Learning techniques to perform Predictive Maintenance tasks in an automated way. The challenge is to use only the alarm counts available in the dataset, without any information about the physical state of the machines. The chosen approach consists of using the alarm counts in a small time window to predict how much plausible it is for a failure to happen the subsequent day.

3.1 Data Preparation

Alarm counts are extracted from the *Alarm records* dataset. A failure instead is set to happen when a maintenance ticket is sent and registered into the *Assistance calls* database. The choice of this dataset rather than the *Operations* one is due to issues occurring in the latter. In particular, maintenance operations are annotated with dates and times which often do not correspond to effective repairs or actions in situ. In fact, a single procedure frequently includes multiple technicians interventions, which span through a variable time window. The dates presented in the database corresponds to the last operation, which may happen several days (if not months) later than the signals sent by alarms. In addition, several maintenance tickets are sometimes grouped together and solved with just one action. On the other hand, help tickets are requested as soon as some malfunction is detected, and hence they provide a more accurate estimate of the failure date and time.

Assistance calls however comprehends annotations that are not connected with faults predictable from alarms. In addition, it contains scheduled repair requests, which are not predictable since they are not tied to alarm escalations nor to equipment degradation. In order to work with a cleaner dataset, the company suggested filtering the help tickets by selecting a list of 80 facilities; among them, maintenance requests are picked if their description contains some keywords related to the cold chain.

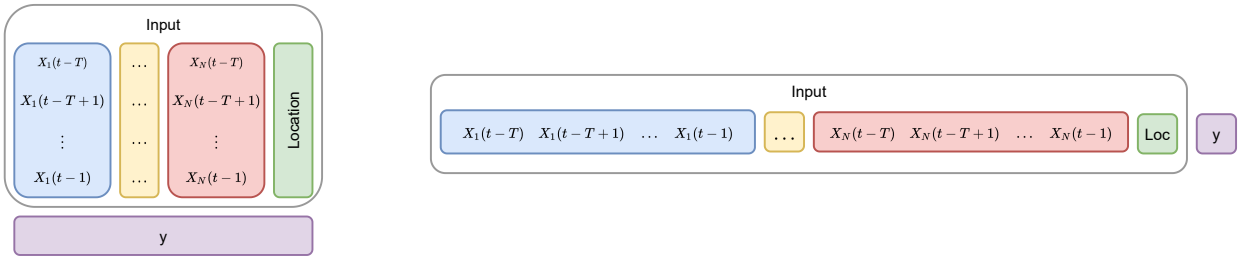
The indications provided by the company suggest that the escalation in the alarms counts that leads to a failure happens rather quickly, in the order of few days. They assure that a time window longer than a couple of days is no significant to predict the faults; moreover, false correlations may arise and can potentially degrade the results. Due to these guidelines, it has been decided to perform an analysis based on the temporal windows' compatibles with the evolution of the problem. For this reason, the time window length is set to be of 3 days in length. As consequence, the input features are constituted by multivariate time series (where each variable is a different alarm ID), with a total length of 3 days. These time series are formed by aggregating together the counts over smaller sub-windows; in practice, aggregation is tested over one-day windows (so each time series is made by 3 time steps, one per day) and over 3 hours windows (which results in 24 time steps, 8 per day).

Each time window is associated with a binary label: 1 indicates that a failure happens the next day, while 0 is the opposite. Input variables and labels are constructed in this way:

1. Alarm counts are grouped together by their ID and facility and are sorted by date and time.
2. They are aggregated over the specified sub-window (1 day or 3 hours). Time series are also filled setting at zero the number of counts of a certain alarm if no entries are found inside the specific window. Hence, at each time step the state of a facility is described by the counts of the alarms ids:

$$
\begin{array}{lc|ccc}
 & \textbf{time} & \boldsymbol{X_1} & \dots & \boldsymbol{X_N} \\
\hline
\text{Facility 1} & t^{(1)}_{\text{start}} & x^{(1)}_{\text{1, start}} & \cdots & x^{(1)}_{\text{N, start}} \\
 & \vdots & \vdots & \cdots & \vdots \\
 & t^{(1)}_{\text{end}} & x^{(1)}_{\text{1, end}} & \cdots & x^{(1)}_{\text{N, end}} \\
\vdots & & \vdots & & \\
\text{Facility M} & t^{(M)}_{\text{start}} & x^{(M)}_{\text{M, start}} & \cdots & x^{(M)}_{\text{N, start}} \\
 & \vdots & \vdots & \cdots & \vdots \\
 & t^{(M)}_{\text{end}} & x^{(M)}_{\text{M, end}} & \cdots & x^{(M)}_{\text{N, end}}
\end{array}
$$

   where $t_{i+1} - t_i$ is constant for every location and X_i are the alarm ids.
3. The facility is encoded as a numeric feature. Calling T the total number of time steps in a time series ($T = 24$ for sub-windows of 3 hours and $T = 3$ for sub windows of 1 day), each element of the dataset can be represented in two ways as the schematic in Fig. 3.1 shows. Description in Fig. 3.1a displays the multivariate time series as a matrix, where each column delineates a different feature, while the rows are the time steps. Categorical features such as the location may be encoded both as a constant vector or as an external variable. This organization for example is required for Recurrent Neural Network Methods. Fig. 3.1b shows the flattened representation of the time series into a one-dimensional vector. This corresponds to a DataFrame and is useful when trying algorithms such as XGBoost which do not usually work with time series.



(a) *This organization is required by models which are designed for multivariate recurrent data (RNN).*

(b) *Time series can be flatten into a single dimension; this allows to use temporal data even with algorithms not specifically designed for them.*

Figure 3.1: *Two different representations of a time series implemented in this work depending on the algorithm chosen.*

4. This procedure applied to the subset of data selected using company indications leads to a very unbalanced dataset. In particular, 99.5% of samples result in class 0 (no failures), while just 0.5% are labeled as 1 and represent fault examples. To try to mitigate this problem, some tests are also conducted generating synthetic data in the training set from the minority class using the SMOTE algorithm (see Sec. 2.2.2.1).

The problem is thus mapped into a binary classification task. Different solutions are proposed in this work; however, results suffer a lot due to the high imbalance of the dataset supplied by the company.

3.2 XGBoost

The first algorithm tested belongs to the random forests group. Boost decision trees are not designed to explicitly deal with time series. Nevertheless, these models can be applied if the data are presented as in Fig. 3.1b, and each time step is treated as an independent variable. The underlying assumption of this solution is that the row counts of alarms provide higher importance than their escalation. That is, faults happen due to sudden changes in the sensors' observations. The implementation is based on the XGBoost variant since it represents the state of the art for BDT.

This approach is tested both aggregating the counts in sub-windows of 1 day and 3 hours in length. The Dataset is split into train (80%) and test (20%) in a stratified way with respect to the label. The training procedure consists of a grid search with a 5-fold cross-validation scheme, resulting in a total of 216 different combinations of hyperparameters. The research is carried on learning rate, the number of tree estimators, maximum depth of the trees, subsample ratio of features and examples given to each estimator, and two types of regularization parameters.

The trees are updated using a binary logistic loss, weighted to account for the class imbalance. Different models instead are compared by their AUC scores. Some runs involve also the use of the SMOTE oversampling technique in the training set to mitigate the disparity, in order to check whether this algorithm can help the model. Eventually, other tests are carried on including the information about the severity of the alarms, which assigns a value of 1, 3, 5, or 10 to each ID. In particular, in one trial counts are weighted by the severity, while the other two experiments feed XGBoost just passing the alarms with severity above a certain threshold.

Table 3.1: *Results of the different XGBoost models tested for failure prediction. Training procedure and parameters tuning involves a grid search with 5-fold cross validation.* Learning Rate*,* Number of Estimators *and* Tree Depth *refers to the best model, and the AUC score on the training set is the average value between the 5 folds. SMOTE oversampling, when applied, is only done on the training set.*

Aggregation	Other Preprocessing	SMOTE	Learning Rate	Number of Estimators	Tree Depth	AUC (train)	AUC (test)
1 day		no	0.1	100	4	0.748	0.714
1 day		yes	0.3	500	8	0.995	0.678
3 hours		no	0.1	100	4	0.736	0.756
3 hours		yes	0.1	500	8	0.994	0.673
3 hours	weight by severity	no	0.1	100	4	0.736	0.755
3 hours	weight by severity	yes	0.3	500	8	0.993	0.643
3 hours	severity > 1	no	0.1	100	4	0.736	0.755
3 hours	severity > 1	yes	0.1	500	8	0.998	0.673
3 hours	severity > 3	no	0.1	100	4	0.736	0.755
3 hours	severity > 3	yes	0.1	500	8	0.998	0.673

Tab. 3.1 displays the results of the study. The first observation indicates a clear difference between runs where oversampling is applied or not. SMOTE leads to better performances on the training set since the discrepancy in the number of samples of the two classes is reduced with respect to the original set, but generalization capabilities are much worse on the test set. In addition, these trials show a model with a higher number of estimators and depth. This means that the complexity is greater, but also leads to overfitting. Secondly, information about severity does not help the model to achieve a better score. The explanation seems that this variable is not related to the failures analyzed in this study. Or else, faults are determined by the combination of all the alarms, rather than being originated just by a smaller subset.

The model outputs a probability for each sample to belong to the label 1 class. In order to assign it to a specific class, a probability threshold p must be decided: every entry with a probability lower than the threshold is set to be a member of group 0, otherwise, the label assigned is 1. Fig. 3.2 shows the confusion matrix at different values of p for the best XGBoost model (which has an AUC score equal to

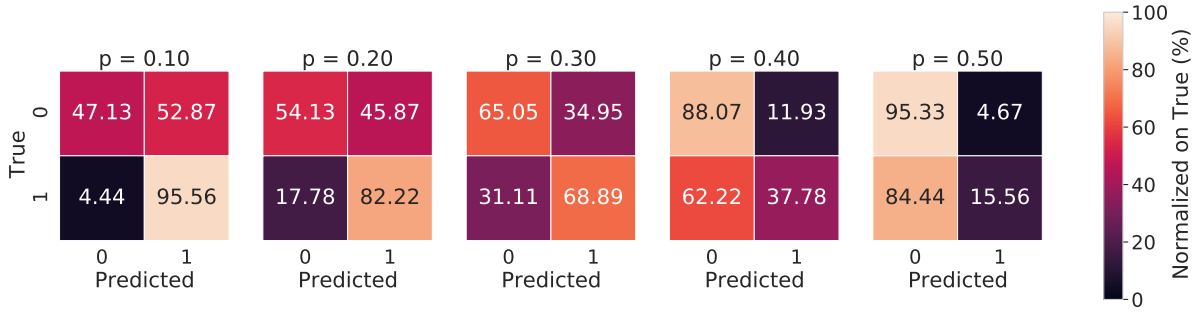


Figure 3.2: *Confusion matrix at different probabilities thresholds for the best XGBoost model. It is obtained using 3 hours aggregation windows, without implementing SMOTE oversampling nor weighting the counts by the alarm severity. Rows represent the true labels, while columns the predicted ones. Values are normalized over the true classes. The AUC score achieved is 0.756.*

Table 3.2: *Metrics at different probability thresholds for the best XGBoost model. f1, Precision, and Recall scores are the average for each label weighted by their support, to account for the imbalance.*

p	Accuracy	Balanced Accuracy	f1	Precision	Recall
0.1	0.472	0.713	0.638	0.997	0.472
0.2	0.542	0.681	0.700	0.996	0.542
0.3	0.650	0.669	0.785	0.995	0.650
0.4	0.879	0.629	0.933	0.995	0.879
0.5	0.951	0.554	0.972	0.994	0.951

0.756). Rows represent the true labels, while columns the predicted ones; results are normalized over the true class. Tab. 3.2 instead displays a set of metrics at the different thresholds; f1, precision, and recall scores are the average for each label weighted by their support, to account for the imbalance.

From these results, it is quite evident that the class imbalance has a strong impact on the metrics. In particular, misclassifying the minority class has a minor impact on the scores. Accuracy, f1, and recall perform better when most of the majority samples are correctly classified, but at the cost of mistakenly attribute the class for most of the minority examples. Balanced accuracy is the only indicator that weighs more the correctness of the small class, rather than the errors in the major one. Confusion matrices show that the model is not able to separate both labels effectively; in order to achieve better results in the minority class, it is necessary to accept a huge degradation in the accuracy. In fact, as in the case with the threshold $p = 0.1$, to classify correctly 95% of the samples with label 1, about half of the prediction in the 0 class become erroneous. BDT solutions usually show good results even in complex scenarios; however, the outcome is not satisfactory in our case. To try to improve them, a different strategy based on recurrent neural networks is implemented in Sec. 3.3.

3.3 Long-short Term Memory Networks

The approach based on the XGBoost algorithm does not take into account the correlations across time between alarms but instead treats each time step as a separate variable. This information can be included by models such as LSTM networks, which are designed for these tasks, as explained in Sec. 2.4.1.

For this approach, the dataset is divided in training and test set in the same way as for the XGBoost setting, in order to have comparable results. Data are aggregated using 3 hours windows. The LSTM is defined using the `Tensorflow` [29] platform. The architecture tested if formed by:

1. Group of 1, 2, or 3 LSTM layers

2. Dropout layer
3. Group of 1, 2 or 3 fully connected dense layers
4. Dropout
5. Single neuron with Sigmoid activation function in order to perform the binary classification and output a probability of belonging to the specific class.

The dropout layers are added to perform regularization during the training. Another method to avoid overfitting is the choice of the Adam optimizer since it implements an adapting learning rate with momentum [30]. The initial value is set to 10^{-3}, as it is the default in the Keras library and the one suggested in the original paper [30].

The training procedure involves grid search using 3-fold cross-validation (the number of folds is reduced with respect to XGBoost due to the larger amount of time required for training). The hyperparameter search is done over the different configurations of the architecture. In particular, the LSTM and fully connected blocks are tested with 1, 2, or 3 layers and a variable number of neuron units; also different dropout fractions are analyzed in this procedure. NNs are trained to optimize a weighted binary cross-entropy loss, and different architectures are then compared using the AUC score.

As before, training is examined both with and without oversampling, but the final performance is always evaluated in the external test set which contains only original entries. Two rescaling approaches are also tried to monitor their impact on the results:

- `MinMaxScaler` scales and translates each feature into the range $[0, 1]$; this transformation may help the learning since it reduces the variance in the data.
- `RobustScaler` is designed for dealing with outliers. Data are centered and scaled by the median and the interquartile range between the 1st and the 3rd quantile, in order to mitigate the effects of mean and variance in the case of outliers.

Table 3.3: *Results of the hyperparameters search using the LSTM architecture. The parameters shown refer to the best model found by the grid search procedure.*

Scaler	SMOTE	LSTM	Dropout 1	Dense	Dropout 2	AUC (train)	AUC (test)
	no	`[50]`	0.5	`[50, 25, 10]`	0.5	0.571	0.481
	yes	`[50,20]`	0.3	`[50]`	0.3	0.915	0.441
`MinMaxScaler`	no	`[50, 20]`	0.3	`[50]`	0.3	0.571	0.471
`MinMaxScaler`	yes	`[50]`	0.3	`[50, 20]`	0.3	0.775	0.376
`RobustScaler`	no	`[50, 25, 5]`	0.3	`[50, 20]`	0.3	0.562	0.491
`RobustScaler`	yes	`[50, 20]`	0.3	`[50]`	0.3	0.922	0.426

Surprisingly, LSTMs perform worse than the XGBoost model, as displayed in Tab. 3.3.

LSTMs performances are shown in Tab. 3.3, and indicate an AUC score below 0.5, which is the score for a random classifier. Again, the use of SMOTE does not help the model; on the contrary, it leads to overfitting, since performances increase in the training data, but drop in the test set. MinMaxScaler does not help the model, but rather leads to worse performances; the explanation could be that data are sparse and present outliers, which degrade the results due to the scaling related to mean and variance. The RobustScaler in fact, which is designed for this kind of setting, provides a slight benefit in the AUC score. Surprisingly, LSTMs results are worse than the ones obtained by the XGBoost model, even though recurrent networks should be able to capture correlations in time, that are not considered by decision trees. Both the model trained without rescaling and with the RobustScaler choose a complex architecture as the best combination of hyperparameters. Without the scaler, there is one single LSTM layer but with a high number of units, and three fully connected ones; in the other case instead, three LSTM layers are required and two dense. These complex architectures suggest

that the model struggles to find correlations between variables and hence it needs a larger amount of neurons. Consequently, a larger amount of data may be required to train the architectures, and this results in the poor performances observed.

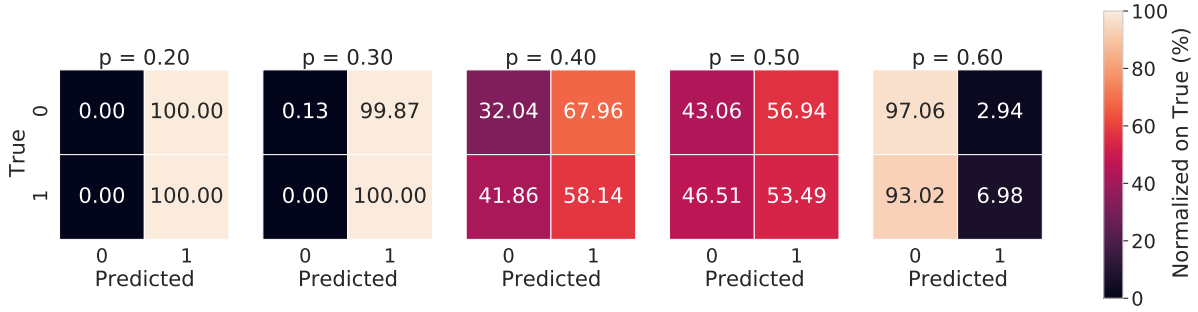


Figure 3.3: *Confusion matrix at different probabilities thresholds for the best LSTM model (the one obtained using the RobustScaler). Rows represent the true labels, while columns the predicted ones. Values are normalized over the true classes. The AUC score achieved is 0.491.*

Table 3.4: *Metrics at different probability thresholds for the best LSTM model. f1, Precision, and Recall scores are the average for each label weighted by their support, to account for the imbalance.*

p	Accuracy	Balanced Accuracy	f1	Precision	Recall
0.2	0.005	0.500	0.000	0.000	0.005
0.3	0.006	0.500	0.002	0.994	0.006
0.4	0.321	0.450	0.482	0.988	0.321
0.5	0.431	0.482	0.597	0.989	0.431
0.6	0.965	0.520	0.977	0.989	0.965

Confusion matrices in Fig. 3.3 point out that most of the probabilities outputted by the network lie in the interval [0.4, 0.5]. The consequence is that the model has difficulties distinguishing the two classes. Below that range instead, all samples are set to the minority class, whereas above the label is predicted as 0. In particular, the results for $p = 0.4, 0.5$ seem to suggests that the behavior is really similar to the one of a random classifier.

The solution based on LSTM brings even more negative results than the XGBoost one. Another approach is then explored during this work.

3.4 NLP-like approach

The third solution proposed is based on the idea that a failure happens due to some specific sequence of alarms, and it is not only correlated to their counts. In other words, knowing if alarm A is triggered before or after alarm B may be more relevant than the information that in the same time interval A is triggered n_A and B n_B times. The idea is to take as an example what is implemented in the Natural Language Processing setting and reframe the problem in a similar way. Hence, the core concept is to build a model able to interpret the "language" which describes the sequence of alarms.

Data preparation is a bit different than what is described in Sec. 3.1. The dataset is grouped by the facility center, and the sequence of alarms is then split into time windows of three days. Each subsequence is then rewritten as a string, where words correspond to the alarms ids (as presented by the schematic in Fig. 3.4). The associated label instead is created as for the other methods: a value of 0 is set if the next day there is no failure, 1 otherwise. It is worth noting that the dataset is still afflicted by the same unbalanced introduced in Sec. 3.1.

Strings are then reduced to the same length by applying padding and truncation; this length value

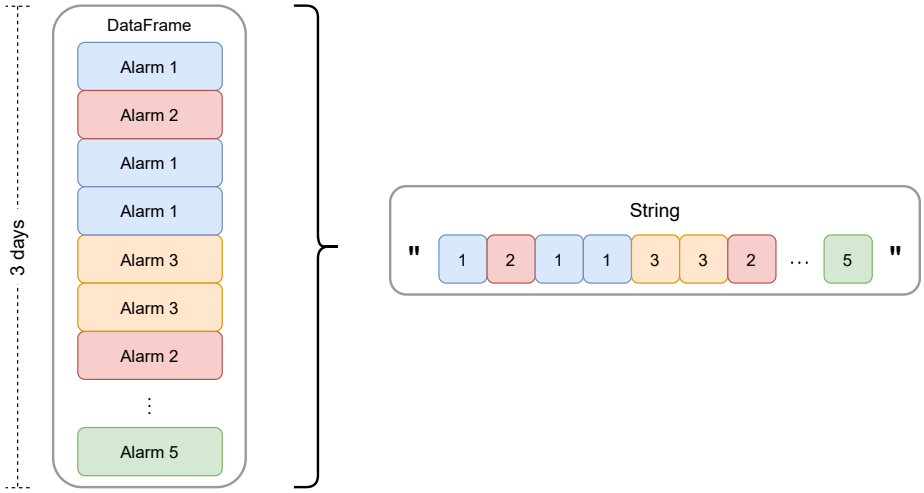


Figure 3.4: *In order to create a language model for the sequence of alarms, each three days sub windows is rearranged as string, where words are represented by the alarms ids.*

is set equal to the quantile at $q = 0.9$ of the distribution of sentence lengths. This choice is done to exclude elements that have abnormal behavior but still keeping as much information as possible without cutting too many entries. In the end, the same split into training (80%) and test (20%) sets as in the previous studies are applied.

The neural network architecture is composed of the elements in Tab. 3.5. The input is mapped into a latent space through an embedding layer; then there two LSTM layers followed by two Dense units; a single Sigmoid neuron makes the prediction for the classification. Dropouts are added to apply regularization during the training.

Table 3.5: *NN architecture implemented for the NLP-like approach to the failure prediction problem. First layer is the embedding one, the output is given by a single units activated by a Sigmoid function. The* Best Configuration *refers to the best model discovered with the Bayesian Random Search, while* Search Space *indicates the distribution chosen from sampling the parameters.*

Layer	**Search Space**	**Best Configuration**
Embedding	Uniform in [10, 150]	109
LSTM	Uniform in [50, 150]	83
Dropout	Uniform in [0, 0.9]	0.36
LSTM	Uniform in [10, 50]	50
Dropout	Uniform in [0, 0.9]	0.25
Dense	Uniform in [75, 200]	188
Dropout	Uniform in [0, 0.9]	0.13
Dense	Uniform in [10, 75]	10
Dropout	Uniform in [0, 0.9]	0.49
Output Neuron with Sigmoid activation function		
Parameter	**Search Space**	**Best Value**
Learning Rate	Log Uniform in $[10^{-5}, 10^{-1}]$	$3 \cdot 10^{-4}$
Batch Size	[8, 16, 32, 64, 128, 256]	32

Hyperparameters tuning involves the size of the embedding layer, the number of units both for LSTM and Dense units, and the fractions of dropout. In addition, also learning rate and batch size are

scanned. Since the number of parameters is considerable, it is chosen to implement a Bayesian Random Search where 64 different combinations of parameters are tested. Every single architecture is trained using a weighted binary cross-entropy loss, and Adam is set as optimizer. Different NNs are then compared using the AUC score. The best model is found to have the parameters in Tab. 3.5; the AUC score in the training set is 0.584, while on the test set is 0.576.

The best configuration shows values that indicate a high complexity, with respect to the defined search space. In fact, layers are chosen to have a high value of units. This indicates the difficulty of the NN to learn the data. This is also reflected in the final AUC score since it is just above the value of 0.5, which denotes a random classifier. This metric is comparatively lower than the one obtained by XGBoost model in Sec. 3.2. However, the performances are found to be better than the LSTM implementation. In particular, the score difference between train and test data is small, which indicates that the model does not overfit, compared to the LSTM one.

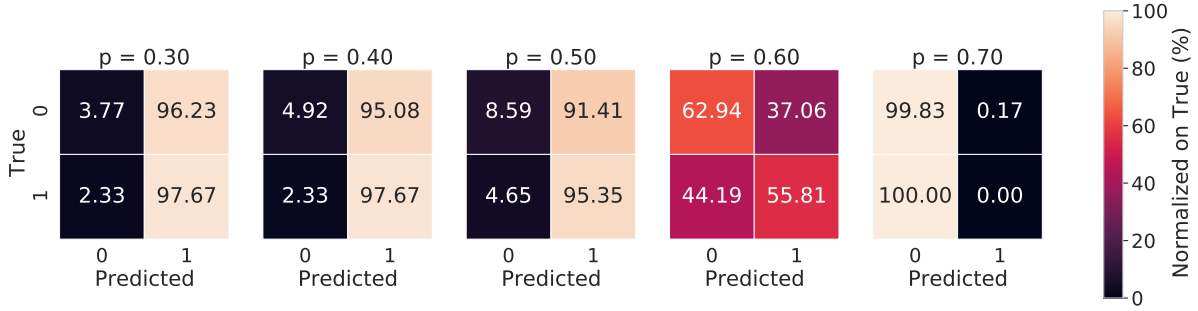


Figure 3.5: *Confusion matrix at different probabilities thresholds for the best NLP-like model. Rows represent the true labels, while columns the predicted ones. Values are normalized over the true classes. The AUC score achieved is 0.576.*

Table 3.6: *Metrics at different probability thresholds for the best NLP-like model. f1, Precision, and Recall scores are the average for each label weighted by their support, to account for the imbalance.*

p	Accuracy	Balanced Accuracy	f1	Precision	Recall
0.3	0.042	0.507	0.01	0.005	0.976
0.4	0.054	0.512	0.01	0.005	0.976
0.5	0.090	0.519	0.01	0.005	0.953
0.6	0.629	0.593	0.02	0.007	0.558
0.7	0.993	0.499	0.00	0.000	0.000

From the study of the confusion matrices, it is pointed out that the critical threshold to separate the class is suggested to be around 0.6 by Fig. 3.5. At this value, half of the minority class is correctly classified, and also 62% of the majority one. However, as in the previous settings, the algorithm still struggles to clearly identify the two labels correctly.

3.5 Ensemble approach: LSTM+XGBoost

It has been observed that the XGBoost approach achieved the highest score among the models proposed. The results in Sec. 3.2 are obtained training it using three days history with three hours time windows, which results in a number of input variables equal to 24 per each alarm ID. This large quantity of information leads the model to learn a very complex setting that in turn may increment the model complexity and then lead to overfitting.

As an attempt to overcome this effect, it is decided to try to train this classifier using the sequence of alarms of the same day instead of the past history (3 days). In this way, the number of inputs is

decreased by a factor of 3 and may lead to better performances. The training procedure uses the same formula as in Sec. 3.2, with a grid search using a 5-fold cross-validation scheme. The AUC score of this model evaluates to 0.844, which is higher than the value obtained in Sec. 3.2. Confusion matrices (Fig. 3.6) and the metrics in Tab. 3.7 indicate that setting the threshold at $p = 0.3$, the accuracy is about 78.9% and the balanced is 0.754; in addition, both classes are correctly classified more than 70% of the times.

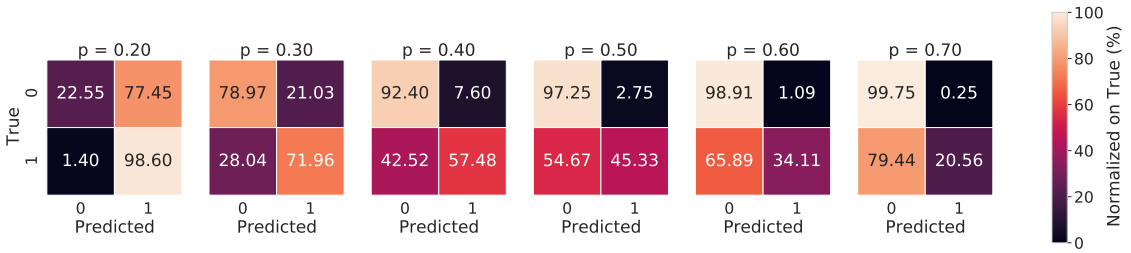


Figure 3.6: *Confusion matrix at different probabilities thresholds for the XGBoost model in the LSTM+XGBoost setup. Rows represent the true labels, while columns the predicted ones. Values are normalized over the true classes. The AUC score achieved is 0.844.*

Table 3.7: *Metrics at different probability thresholds for the XGBoost model in the LSTM+XGBoost setup. f1, Precision, and Recall scores are the average for each label weighted by their support, to account for the imbalance.*

p	Accuracy	Balanced Accuracy	f1	Precision	Recall
0.2	0.229	0.605	0.366	0.994	0.229
0.3	0.789	0.754	0.877	0.992	0.789
0.4	0.922	0.749	0.954	0.992	0.922
0.5	0.969	0.712	0.980	0.992	0.969
0.6	0.985	0.665	0.988	0.992	0.985
0.7	0.993	0.601	0.992	0.992	0.993

These promising results motivated to define a new approach incorporating an LSTM to predict alarm counts in the next day given the past history, and XGBoost to make the binary classification. A schematic illustration of the approach is provided in Fig. 3.7.

To train the network, data is prepared as described in Sec. 3.1, using sub-windows of three hours for aggregation. However, the output to predict is not a binary label; instead, also the counts of the next day are organized as a time series, where the counts are again aggregated in three hours ranges. More precisely, the input is composed by the alarms counts in 8 timestamps per day times 3 days for each alarm ID, while the output are the ones in the 8 time steps of the consecutive day.

The chosen architecture features a simple structure: input is passed to an LSTM layer, followed by a Dense one with a dimension of 8 (number of output time steps) times the number of alarms. The result is then reshaped as a matrix of shape 8× the number of ids. Hence, the architecture is a simple one-shot model, which means that all of the outputs are calculated in one single step. The training of the model is performed adopting the mean absolute error (MAE) between the predictions and the true values as loss and Adam as optimizer. The final model displays an MAE of 0.032 and MSE equal to 1.055.

To measure the performances of the overall model, LSTM is applied to the whole dataset to produce an estimate of the alarm counts of the next day given the three past days' time series. The time series predicted by the LSTM are used as inputs to an XGBoost model, which performs the binary classification task between the presence or not of a failure. The final AUC score indicates a value of 0.660, which is however lower than the one produced by simply applying XGBoost algorithm, thus

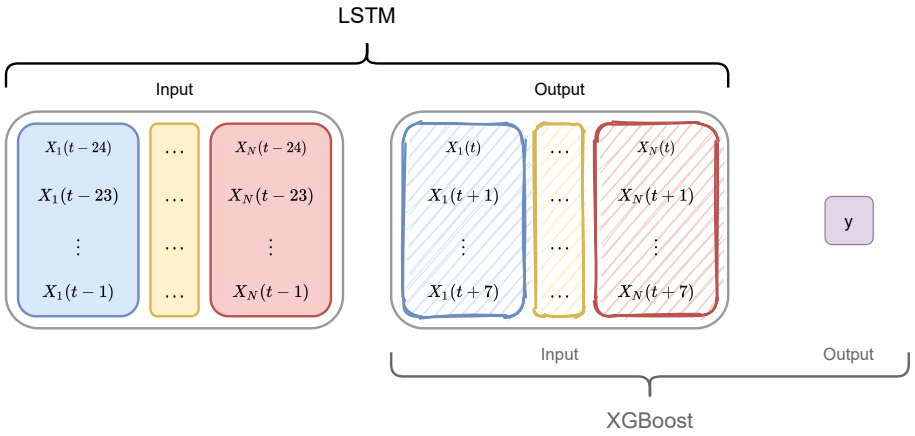


Figure 3.7: *LSTM network is trained to predict the alarms fro the next day from the time series of the past three days. XGBoost uses this information to perform the binary classification and detecting whether a failure occurs.*

pointing to worse performances with respect to the simple XGBoost approach discussed in sec in Sec. 3.2.

Comparison between Fig. 3.8 and 3.6 and the metrics in Tab. 3.8 and 3.7, suggests that the LSTM part is the week point of the pipeline. In fact, classification performances highly decrease with respect to the single XGBoost. In addition, the overall architecture is once more not able to achieve the same good results for both classes. At $p = 0.3$, which for the XGBoost classifier alone yields more than 70% accuracy for both labels, the minority group is not individuated 63% of the times. This observation remarks the problem of the class imbalance, which makes it difficult to learn both classes.

3.6 Results

This Chapter provides some tentative solutions to tackle the problem of predicting failures in this specific use case. Among the four different approaches proposed, better results are achieved by the XGBoost model, with an AUC score of 0.756. Tab. 3.2 suggests that setting a probability threshold of 0.3 gives the opportunity to identify a failure about 70% of the time, at the cost of interpreting 35% of normal activities as faulted. Lowering the threshold, an accuracy of 95% can be reached for the minority class, but half of the times failures are detecting even if machines are not broken. The choice of being more or less conservative must consider a trade-off between accepting the risk of unexpected faults, and the costs due to maintenance operations. Choosing a lower value of p means that all failures are predicted, but there are also lots of "useless" interventions, in the sense that technicians need to get involved even if they should not be required. On the other hand, a higher value shifts the risk towards the Run-to-Failure approach: useless operations are avoided, but failures are also never spotted.

The other solutions based on NN do not produce satisfactory results. LSTM network behaves similarly to a random classifier (in fact the AUC score is below 0.5). In particular, it is not able to learn the data. Tab. 3.3 shows that the parameters search leads to a pretty big model, which may be too complex to be trained with such an imbalanced dataset.

The NLP-like method tries to reshape the problem as in the context of Natural Language Processing. Results however are similar to LSTMs. The AUC score is 0.576, which labels this algorithm almost unreliable as a random estimator. Fig. 3.5 indicates that at thresholds different from 0.6, all samples are classified in the same class. Hence, the probability score outputted by the network does not exhibit a large gap between samples of class 0 and 1: the result is that the model struggles to identify the two labels.

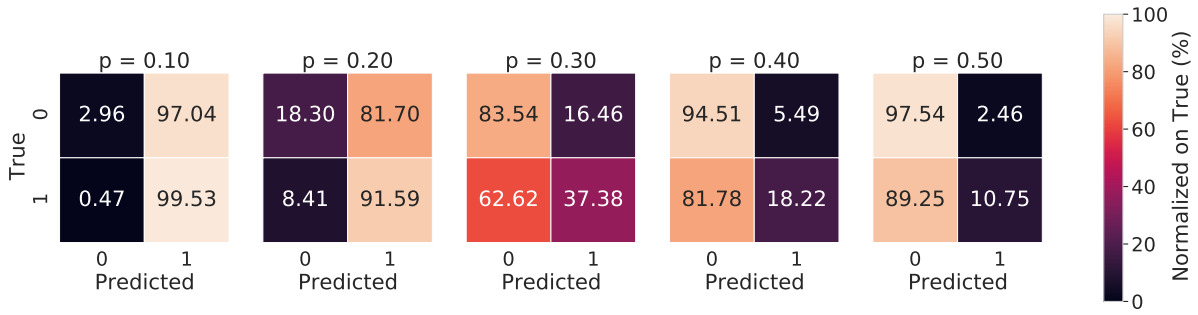


Figure 3.8: *Confusion matrix at different probabilities thresholds for the best NLP-like model. Rows represent the true labels, while columns the predicted ones. Values are normalized over the true classes. The AUC score achieved is 0.660.*

Table 3.8: *Metrics at different probability thresholds for the best NLP-like model. f1, Precision, and Recall scores are the averages for each label weighted by their support, to account for the imbalance.*

p	Accuracy	Balanced Accuracy	f1	Precision	Recall
0.1	0.034	0.512	0.057	0.993	0.034
0.2	0.186	0.549	0.307	0.992	0.186
0.3	0.833	0.604	0.904	0.990	0.833
0.4	0.941	0.563	0.964	0.990	0.941
0.5	0.970	0.541	0.980	0.990	0.970

Finally, a pipeline composed of an LSTM and an XGBoost is analyzed. The neural network is responsible for predicting the future of the time series the next day, given the previous three-day history. The decision tree instead, classify every single day. XGBoost alone shows very good performances (see Tab. 3.7). The confusion matrix at $p = 3$ in Fig. 3.2 shows an accuracy above 70% in the classification for both classes. However, the performances deteriorate when applied to the prediction of the NN; at the same threshold, the minority class is misclassified 60% of the time.

Results of these studies clearly show a struggle due to the high imbalance in the dataset. All the models present difficulties in achieving high accuracy in locating both classes simultaneously. The fact that probabilities assigned to class 0 and class 1 are not divided by a large gap (as shown by the confusion matrices resulting from the models), is a sign that the task is very complex even using non-linear methods like decision trees and neural networks.

The majority class having 99.5% of the total samples also afflicts the metrics scores. This disparity is caused by the company suggestions in filtering the *Assistance calls* database. In particular, the problem may reside in the hard-coded words requested to mark events as correlated to cold cycle faults. To overcome this problem, in Chapter 4 different algorithms are explored in order to automatically extract topics from the assistance calls.

4

Natural Language Processing for Ticket Analysis

Results achieved in Chapter 3 clearly indicate that the unbalance in the dataset negatively influences the outcomes of the proposed models. In particular, models struggle to detect the minority class, which corresponds to the presence of failures. Since the cold cycle faults were extracted by requesting some hard-coded words to be present in the maintenance tickets, there is the possibility that lots of details are lost. This Chapter explores a new approach to independently extract information from the logs of repair interventions and maintenance requests.

Selecting events by requesting specific keywords limits the results since it does not clearly define a topic or argument. The idea is then to perform an unsupervised topic clustering and see if it can discover similarities between documents (Sec. 4.2). Sec. 4.3 instead focuses on a different approach. A model is trained in a supervised setting using the Failure ID as the class label, and then, apply the trained algorithm to classify the other tickets.

4.1 Data Preparation

Data used in this analysis come from *Assistance calls* and *Operations* files (see Sec. 1.3). In particular, the field containing the description of the request/intervention is extracted and studied. Since this data are of string type, text preprocessing techniques are implemented.

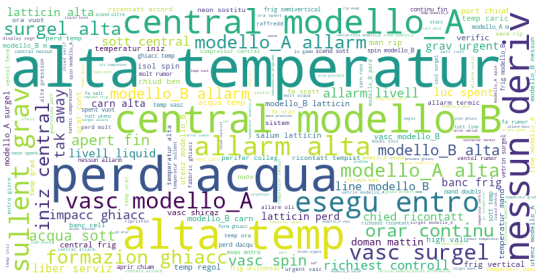


Figure 4.1: *Word Cloud visualization of the most common words in the dataset.*

First of all, the text is converted to lower case, and punctuation is trimmed away; also every character not belonging to alphabet letters is deleted. Stopwords (such as articles, prepositions, ...) are then removed from the phrases (using `Gensim` package [31]). In addition to them, some other terms are cut out: these were chosen by inspecting the results of analysis such as the one described in Sec. 2.5.1 using interactive tools as LDAvis [32], implemented in the package `pyLDAvis`. These additional words were common in the data, but did not bring useful information to classify topics such as: *buongiorno* (good morning), *grazie* (thanks), *saluti* (greetings), ... Lastly stemming is also applied. It is the process to reduce inflected (or sometimes derived) words to their word stem, base, or root form. For examples, verbs are reduced to their infinite form (e.g. *mangio* becomes *mangiare*), and nouns such as *signore*, *signori*, *signora* are set to their root *signor*.

Fig. 4.1 and 4.2 show visually the distribution of the most common words. The words "modello_A" and "modello_B" refers to two different types of machines; their model acronym has been anonymized

since they are proprietary to the company.

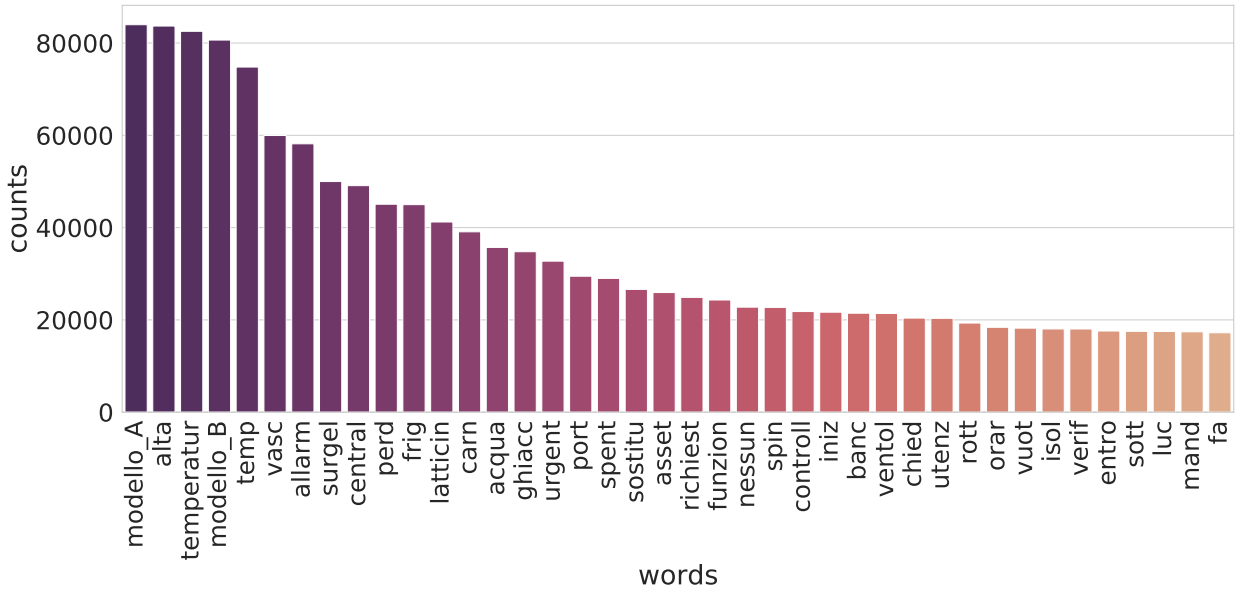


Figure 4.2: *Histogram of the most frequent words in the preprocessed dataset.*

4.2 Unsupervised topic clustering

In order to obtain relevant tickets from the *Assistence calls* dataset, it has first performed an unsupervised topic clustering. The goal is to extract sentences which can be associated to the semantic area described by words such as: *ghiaccio* (ice, frost), *temperatura* (temperature), *impaccamento* (packing), and so on. In this way faults related to the cold chain may be isolated and grouped together, while events not linked to the alarm signals (such as light bulb substitutions, planned interventions, ...) should be pruned out. Latent Dirichlet Allocation and Doc2Vec models are tested for this scope.

4.2.1 Latent Dirichlet Allocation

Latent Dirichlet Allocation (LDA) is a generative statistical model, which extracts topics from document based on the words in them (see Sec. 2.5.1).

The algorithm is tuned by applying a grid search technique with a 4-fold cross validation procedure on the learning rate, number of iterations and number of topics. The input of the model consists on the document-word matrix created by `sklearn CountVectorizer` class. Given the list of N documents (the maintenance tickets in this case), it builds a vocabulary with all the M words present in them; then the document-word matrix is an $N \times M$ matrix, where the element (i, j) indicates how many times word j appears in document i. The results of the grid search are displayed in Fig. 4.3, where the best performances have been obtained using the following list of parameters:

- learning rate = 0.9
- number of iterations = 45
- number of topics = 5
- Score (log likelihood) = $-6.72 \cdot 10^{-6}$
- Perplexity = 722.

It is clear that the number of topics impacts severely the final score of the LDA; in particular, the great degradation of performances with the increase in the number of topics indicates that there may

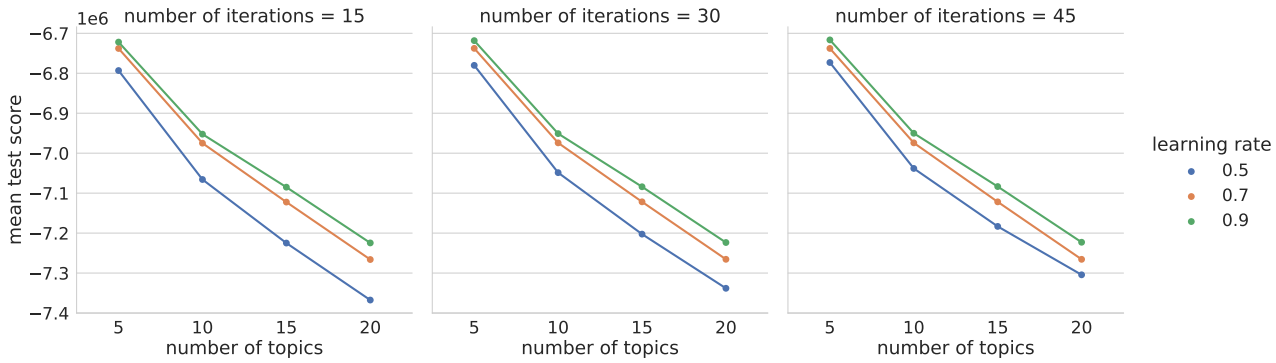
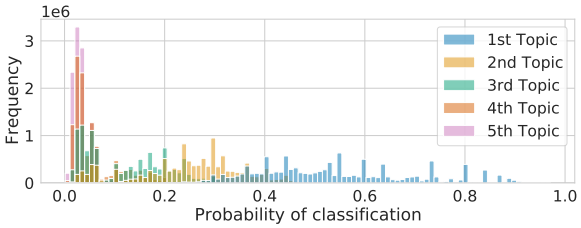


Figure 4.3: *Cross validation score for the Latent Dirichlet Allocation parameter grid search. Best configuration is found to have: (i) learning rate* $= 0.9$ *(ii) number of iterations* $= 45$ *(iii) number of topics* $= 5$*.*

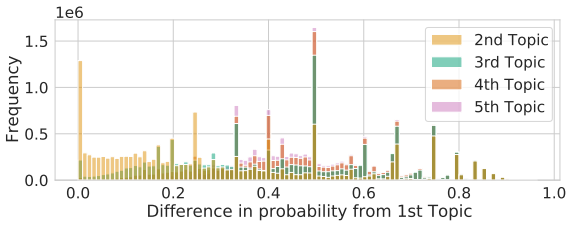
be a difficulty in defining the topics. The best configuration described above is also trained using the TF-IDF feature matrix returned by `sklearn TfidfVectorizer`. This class weights the term-frequency (TF) matrix by the inverse document-frequency (IDF), which for a term t corresponds to:

$$\mathrm{IDF}(t) = \log\left(\frac{1+N}{1+\mathrm{DF}(t)}\right) + 1\,, \tag{4.1}$$

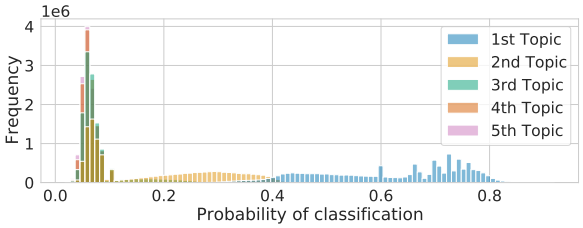
where N is the total number of documents, and $\mathrm{DF}(t)$ is the number of documents that contain the word t. The resulting vectors are then normalized by the Euclidean norm. This further preprocessing is not mandatory for LDA, but it may help the algorithm in the construction of the dictionary by assigning a lower weight to frequent terms, which may not carry significant information.



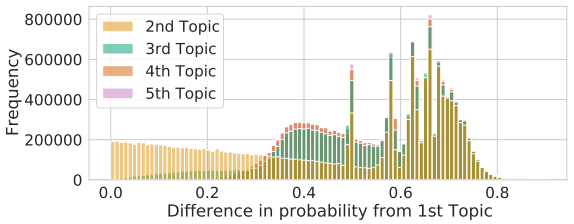
(a) *Distribution of the probability values of being assigned to a topic, from highest to lowest, for the document-word LDA. 1st Topic indicates the topic with the higher probability, while the 5th is the one less probable.*



(b) *Distribution of the difference in probability between the most probable topic and the others for the document-word LDA.*



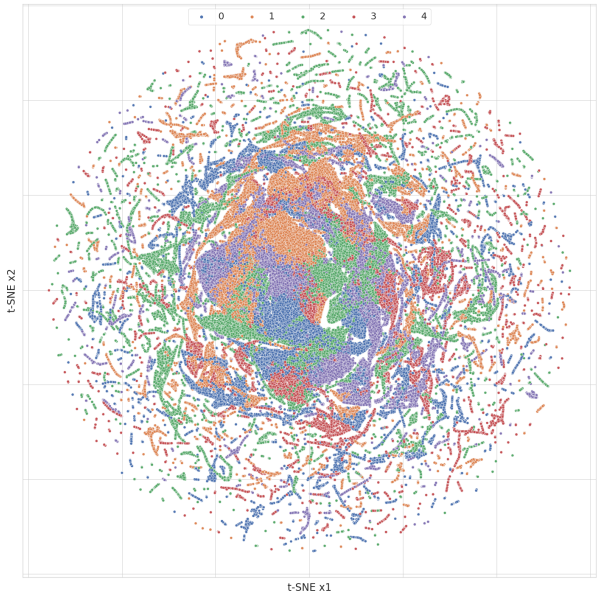
(c) *Distribution of the probability values of being assigned to a topic, from highest to lowest, for the TF-IDF LDA. 1st Topic indicates the topic with the higher probability, while the 5th is the one less probable.*



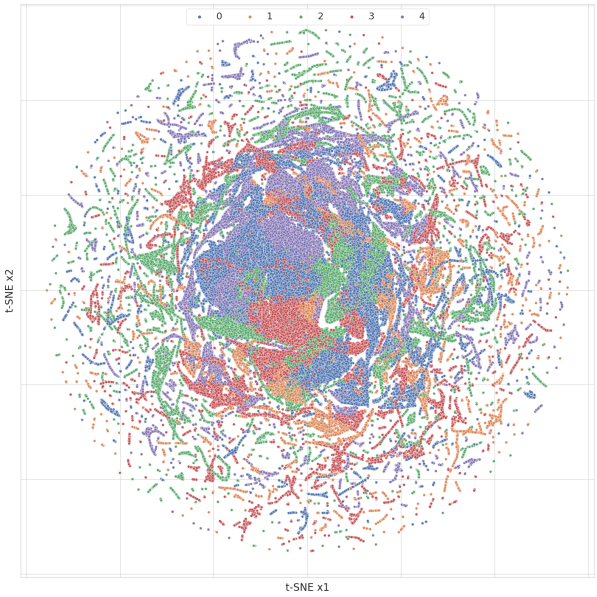
(d) *Distribution of the difference in probability between the most probable topic and the others for the TF-IDF LDA.*

Figure 4.4: *Distributions of the probability of belonging to a topic for LDA. Top row represents results using the document-word matrix, while the bottom one TF-IDF features. Plots on the left compare the probability of being assigned to each of the 5 topics, from the most probable (*1st Topic*) to the least one (*5th Topic*). Right figures represent the difference in probability between the most probable topic and the others.*

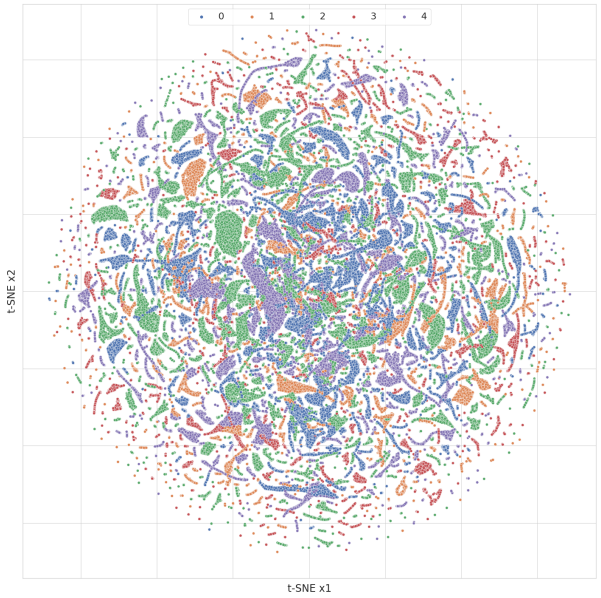
One of the major drawbacks of LDA algorithms resides in the struggle to understand the results. For each document, this algorithm assigns a probability of belonging to every of the identified topics.



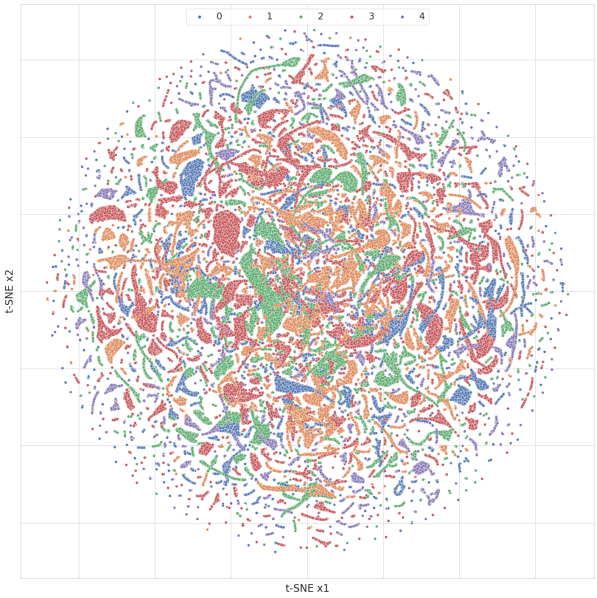
(a) *Document assigned to the topic with highest probability (document-word LDA).*



(b) *Document clustered using k-means algorithm setting k equal to LDA number of topics (document-word LDA).*



(c) *Document assigned to the topic with highest probability (TF-IDF LDA).*



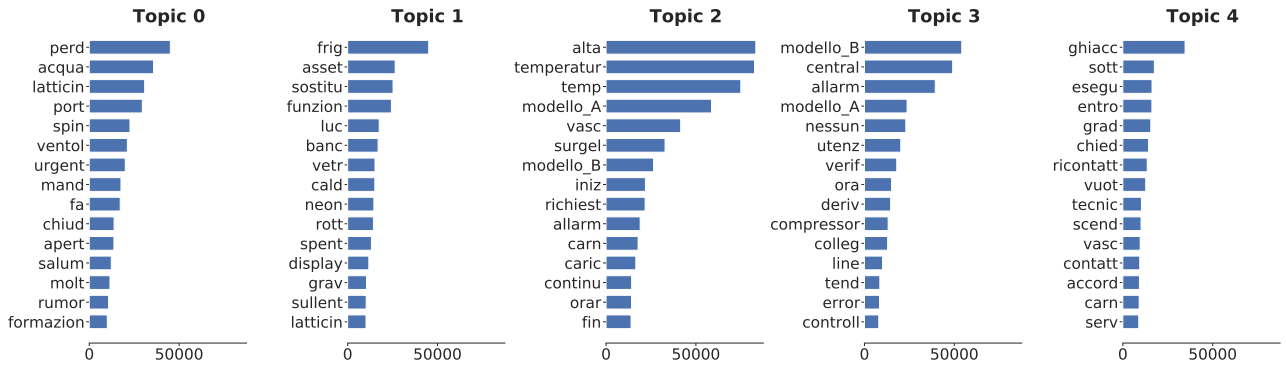
(d) *Document clustered using k-means algorithm setting k equal to LDA number of topics (TF-IDF LDA).*

Figure 4.5: *Clusters created by LDA. Top row shows results using document-word matrix, bottom one with TF-IDF features. Graphics on the left column assign each document to the topic with highest probability, while on the right clustering is performed through k-means technique on the probability scores.*

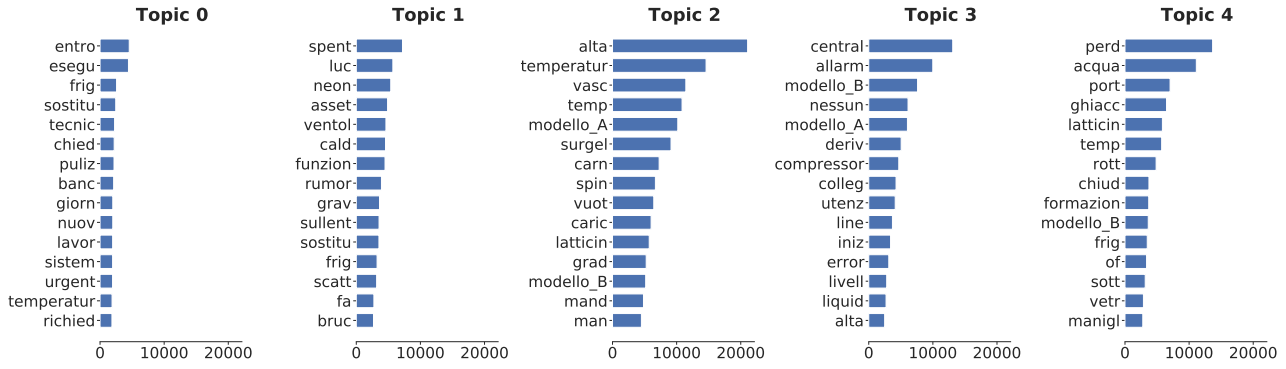
Fig. 4.4 shows the probability distributions returned by the LDA. In plots 4.4a and 4.4c, topic probabilities are ordered from higher to lower, so that the *1st Topic* is the most probable one for the given document. It is possible to notice that the first topic displays a distribution shifted towards the range $[0.4, 1]$, whereas the other ones are below 0.4. This may tell that the algorithm is quite certain shows the probability distributions returned by the LDA. However, graphics 4.4b and 4.4d indicate that the difference between the most two probable topics is not always very large, but instead sometimes it is lower than 0.2. This confusion is particularly pronounced by the model fed with the document-word matrix, as it can be observed in Fig. 4.4b.

In order to visualize if the topics assigned do actually form clusters of consistent documents, t-SNE is applied to the probabilities drawn by LDA to reduce the number of dimensions from 5 to 2. This method should preserve the relations between points, hence if clusters are present, they should be visible by this representation. Fig. 4.5 displays t-SNE results, both for LDA instantiated with document-word matrix (top) and TF-IDF (bottom); topics are further assigned either by choosing the one with highest LDA probability or by an independent k-means clustering on the LDA features. It is clear that well-defined clusters are not present; hence this algorithm struggles to assign similar features to documents belonging to the same group.

In addition to that, LDA does not provide a clear way to understand which is the connection between the words inside the documents and the topic assigned to them. Some level of understanding of this mapping might be extracted by analyzing the most relevant words for each topic. The importance of the first 15 words for each topic is displayed in Fig. 4.6. In Fig. 4.6a, Topic 2 seems to be correlated with faults related to high temperatures in the machines (*alta*, *temperatur*, *temp* are the top 3 terms), while Topic 4 assigns an high weight to the word *ghiacc* (ice). The model fed with TF-IDF features (Fig. 4.6b) displays again the presence of *alta*, *temperatur*, *temp* (which indicates high temperature in the systems) in Topic 2, and Topic 4 gathers together *perd*, *acqua* (water loss) and *ghiacc* (ice). However, it is not easy to extract the clusters which are connected to faults in the cold chain.



(a) *LDA fed with document-word matrix.*



(b) *LDA fed with TF-IDF features.*

Figure 4.6: *Most relevant words for each topic discovered by LDA decomposition.*

Summing up, the difficulties in splitting documents into topic clusters and the problems associated with their semantic interpretation suggest that LDA is not suitable for our purpose.

4.2.2 Doc2Vec model

The second approach is based on Word2Vec and Doc2Vec. The idea is to group documents based on the features extracted by those algorithms, and then look at the vector representation of their means in order to extract words that describe those clusters.

A Doc2Vec model is trained over the dataset using the Gensim library. Each ticket is mapped into a 100-dimensional vector, which corresponds to the embedding dimension of the model. Clusters are then retrieved by applying a k-means algorithm over the document embedded representation, using cosine distance to measure them, since it is the metric implemented Doc2Vec and Word2Vec models to measure the semantic similarity (see Sec. 2.5.2). The number of clusters k is set equal to 5, in order to be consistent with the LDA approach.

The results are visualized in Fig. 4.7, where a PCA is first implemented to reduce the number of features from 100 to 10, and then a t-SNE is performed on the principal components to obtain only the two coordinates of the plot. The use of PCA is required in order to be able to run the other algorithm in a reasonable amount of time. The representation learned from Doc2Vec is not able to split documents into clear clusters: with the exception of the documents assigned to the green k-means cluster in Fig. 4.7, all the other categories form just one big blob, which indicates that the model is not able to discriminate between them.

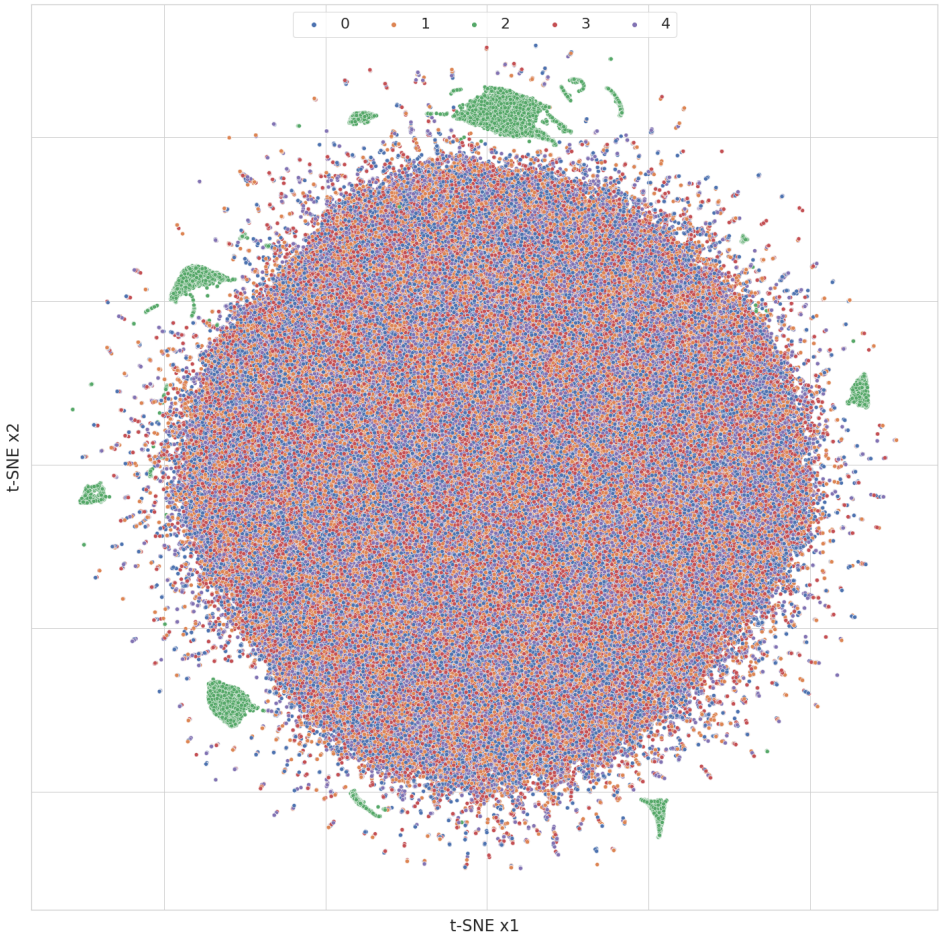


Figure 4.7: *Document embedding learnt by Doc2Vec model with the cluster assigned by k-means. Vectors representing documents lives in a 100-dimensional space; to plot them first a PCA is applied to reduce them to 10 components and then a t-SNE is performed.*

Table 4.1: *Most frequent words for each cluster discovered using Doc2Vec model.*

Cluster 0	Cluster 1	Cluster 2	Cluster 3	Cluster 4
temperatur	modello_A	alta	alta	modello_A
alta	alta	modello_A	modello_A	temperatur
modello_A	temperatur	modello_B	temperatur	alta
modello_B	modello_B	temperatur	modello_B	modello_B
temp	temp	vasc	temp	temp
vasc	vasc	allarm	vasc	vasc
allarm	allarm	surgel	allarm	allarm
central	surgel	mand	central	central
surgel	central	temp	surgel	surgel
perd	frig	iniz	perd	perd
frig	perd	carn	latticin	frig
latticin	latticin	richiest	frig	carn
carn	ghiacc	frig	carn	latticin

This observation is also confirmed by studying the most common words in each cluster (Tab. 4.1). All the clusters share more or less the same words: *temperatur/temp* (temperature), *alta* (high), *alarm* (alarm), ...

It can be argued that the problem may reside in the fact the corpus is not large enough to train such a complex model. To overcome this issue, a test has been performed by adopting a pre-trained network. The chosen model is BERT, as it represents the state of the art for NLP tasks (Sec. 2.5.3). In particular, the `bert-base-italian-xxl-uncased` [33] variant is used, which has been trained over a corpus of size equal to 81 GB, containing text from Wikipedia, OPUS corpora [34], and OSCAR corpus [35]. The model is available through the `Hugging Face` [36] library, which provides the state of the art models in the field of NLP.

BERT maps documents into vectors in a 768-dimensional embedded space. As for Doc2Vec, k-means clustering is then performed using the cosine distance, in order to group together items that should share the same topic. Results visualization is available in Fig. 4.8, where the dataset dimensionality is first reduced to 10 by a PCA, and then t-SNE maps these coordinates into two dimensions to plot them, analogously to what previously reported for the Doc2Vec results in Fig. 4.7.

With respect to the previous Doc2Vec model (Fig. 4.7), documents appears to be arranged in a smoother way. In particular, points inside blue, red, and green clusters occupy near regions in the plots, in contrast with the results of the Doc2Vec model. Therefore, a pre-trained model such as BERT is able to encode better the tickets than a simpler Doc2Vec model trained from scratch and directly on the available ticket dataset itself. Nevertheless, problems arise when trying to assign a topic to each cluster. As for Doc2Vec, Tab. 4.2 displays that all groups share the the same words as the most frequent ones: *model_B*, *model_A*, *temperatur* (temperature), *alta* (high), ... This means that even though the vector representation of documents achieves a better mapping, the algorithm is still not able to detect different semantic areas that may help to understand which entries should be considered to train the models for predictive maintenance.

In conclusion, as for the LDA method, the Doc2Vec approach does not guarantee reliable topic classification in this unsupervised scenario. Better performances seem to be reached by the BERT algorithm, even though its results cannot be clearly connected to specific words which can identify failures connected to the cold chain.

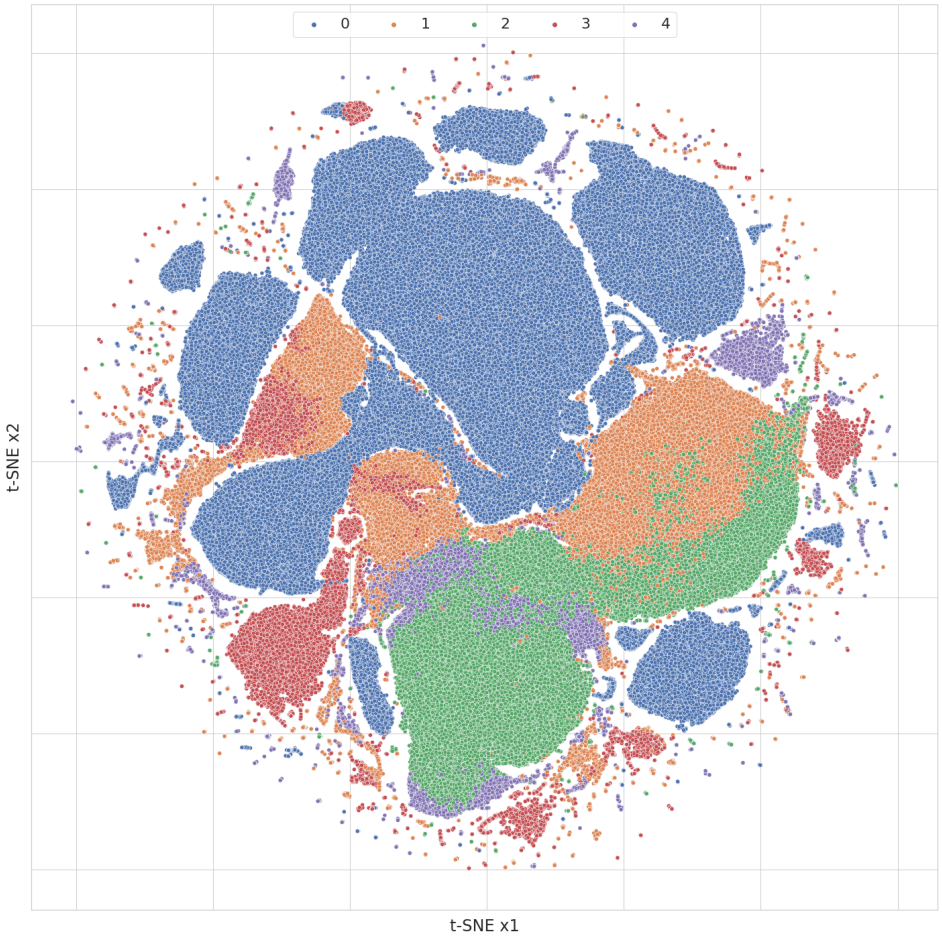


Figure 4.8: *Document embedding learnt by BERT model with the cluster assigned by k-means. The embedding resides in a 768-dimensional space. To represent these points, the dimensions have been reduced to 10 using a PCA, and then a t-SNE is applied to obtain two coordinates.*

Cluster 0	Cluster 1	Cluster 2	Cluster 3	Cluster 4
modello_B	modello_A	temperatur	modello_A	alta
temperatur	alta	modello_B	perd	temperatur
modello_A	temp	alta	acqua	latticin
temp	modello_B	temp	temp	central
alta	temperatur	modello_A	alta	perd
allarm	surgel	vasc	surgel	nessun
vasc	allarm	carn	vasc	temp
frig	central	central	modello_B	deriv
surgel	vasc	perd	port	modello_B
central	latticin	allarm	latticin	acqua
urgent	perd	surgel	ghiacc	allarm
ghiacc	ghiacc	acqua	sostitu	spent
carn	carn	urgent	carn	sostitu
perd	funzion	ghiacc	funzion	entro
richiest	sostitu	spent	central	esegu

Table 4.2: *Most frequent words for each cluster discovered using BERT model.*

4.3 Supervised Topic Classification

Unsupervised topic clustering does not bring satisfactory results. Models seem not to be able to split tickets accordingly to their contents, and their results are obscure and difficult to interpret. As an attempt to overcome these problems it has been decided to exploit the *Operations* dataset. As described in Sec. 3.1, its content is not useful to predict failures, since information reported in this document is assigned with dates later in time than the malfunctions. However, since it is compiled by technicians after they solved the issue, the fault is clearly reported. In this way, it is possible to associate the description of a failure with a flag given by technicians after the repair.

The paradigm is then shifted into a supervised learning problem, where a label is assigned to a text. If the model manifests good performances, it could be exploited to classify the tickets of Sec. 4.2; then going back, a more meaningful dataset could be extracted and the models proposed in Chapter 3 may produce better results if retrained.

4.3.1 SpectrumBoost and BERT models

The text is prepared as described in Sec. 4.1. However, since results from the unsupervised analysis show the difficulty of splitting tickets into topics, not all entries are used. In fact, as displayed in Fig. 4.9, among the 55 different classes of failures, only a small number contains a good amount of observations. In order to avoid a strong class imbalance, only the top 3 IDs with respect to their frequency are chosen:

1. ID = 105 is a fault related to the cold cycle.
2. ID = 115 indicates failures correlated to the electric implant.
3. ID = 140 refers to broken body parts.

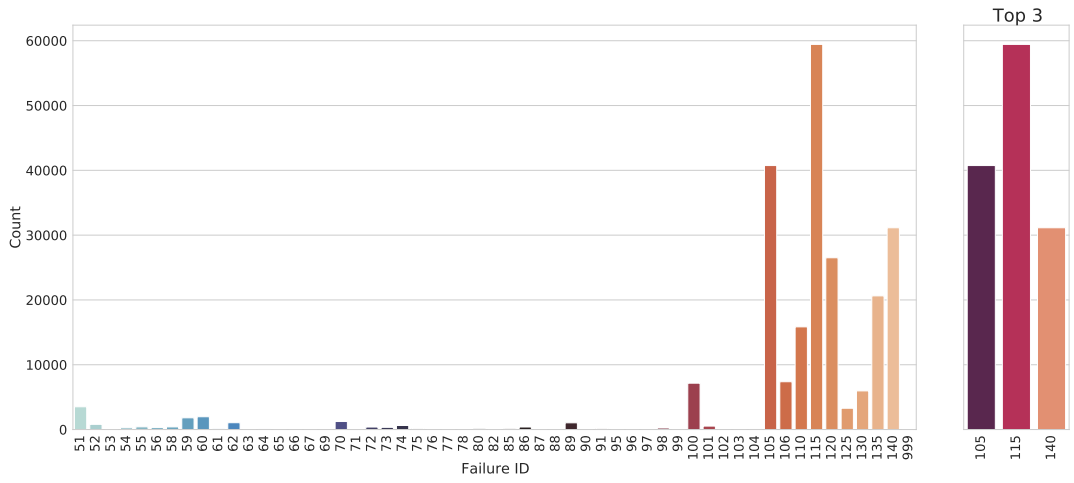


Figure 4.9: *Frequencies of the failure IDs reported after technicians operations. The right histogram shows the three most frequent classes, which are the one chosen to train the classifier.*

To solve this classification problem, two main approaches are implemented.

SpectrumBoost (Tab. 4.3) Spectrum kernel of order p extract features that are related to the number of occurrences of a sub-string with a fixed length p (see Sec. 2.5.4). In this way, the focus is on local information rather than long-term dependencies inside the document. This approach is taught to be functional, since maintenance requests are generally short pieces of text and the meaning is enclosed into the repetition of some keywords (such as *ghiaccio* – ice –, *temperatura* – temperature –, ...), instead of being distributed into long descriptions. In addition, looking at sub-strings helps to mitigate spelling errors in a similar way as a stemmer. For example the word ice may be written as *ghiaccio*, *ghiacci*, *ghiacc*. These terms may be

interpreted as different if the model maps each word into a unique vector, while algorithms such as the spectrum kernel may be able to extract common features.

Spectrum feature extraction is implementing setting $p = [3, 4, 5, 6, 7]$. In order to compute the kernel matrix, Nystrom approximation is required to avoid out-of-memory errors. The order of the approximation is set to 5000 since it is the maximum achievable by the hardware resources available (110 GB of RAM). This means that from each document, 5000 features are extracted. As shown in Tab. 4.3, this passage is both hardware intensive and time-consuming, since it requires about one hour using optimizing algorithms for advanced algebraic operations on GPU. Features are extracted both by the stemmed and non-stemmed version of the dataset.

An XGBoost model is then trained to perform the classification using the variables extracted by the spectrum kernel. The dataset is split into train and test sets (respectively 80% and 20% of the samples). The training part is further sub-divided into train (90%) and validation (10%). A parameter tuning procedure is carried out with a random Bayesian search over 7 different hyper-parameters, analyzing 64 different possible configurations. The maximum number of epochs (that is actually the number of estimators in the context of gradient trees) per try is set to 1500; however, early stopping is implemented by monitoring the logarithmic loss of the validation set, with patience of 20 epochs. The Bayesian optimization is performed by studying the f1 score on the validation set. The parameters tuning is run in parallel on two GPUs. Finally, the best configuration is then retrained, and the time required is annotated in the Table. Performances are evaluated by f1, AUC, balanced accuracy, and raw accuracy metrics on the test set.

In addition to single spectrum features, two test combining different values of p are provided. In the first one, variables from spectrum $p = 3, 4, 5$ are summed element-wise; the second instead makes use of the 1000 most important features (ranked by XGBoost models) for each spectrum kernel.

BERT models (Tab. 4.4) The second group of algorithms consists of BERT variations. They represent state-of-the-art in the field of NLP, and their architecture is based on transformers (see Sec. 2.5.3). Different pre-trained networks are tested (more information about the training corpus for each variant is provided in the notes in Tab. 4.4).

The same split of train, validation, and test as for SpectrumBoost is applied to the dataset. One dense layer is appended to each pre-trained architecture and the models are fine-tuned for this specific task. As the domain of the *Operatiions* dataset is quite specific, it is decided not to freeze the weights of the model to their pre-trained values. Instead, the whole model has instead been retrained. More precisely, this procedure is not a proper fine-tuning, but rather a training with a "warm" start. The training is carried out without defining a number of epochs, but defining an early stopping with patience of 5 iterations which monitors the f1 score over the validation set. Since this task is really demanding in terms of resources and time, and due to the complexity of the networks, no hyperparameter search is implemented, and the default values of the BERT architectures are used. Performances are then assessed on the test set, in order to compare the two approaches.

The results of these two different procedures seem to head in the right direction. Accuracy (both raw and balanced) is above 70%; spectrum kernel method provides higher raw accuracy (best result indicates a score of 72.8%), while BERT performs slightly better on the balanced version (although dropping on the standard one by 1%). Since some residual imbalance can be observed between the 3 selected classes of the dataset, AUC and f1 metrics deliver important information to understand performances. Both criteria suggest that SpectrumBoost behaves better than BERT, as the first solution provides a gain of about 1% with respect to the latter.

The outcome is quite interesting since the state-of-the-art solution shows the worst performance in this task compared to SpectrumBoost. One possible explanation resides in the specificity of the domain of the dataset, which is related to a very peculiar area of interest. BERT architectures are trained on huge corpora, but there is no focus on such a specific topic as the one in this task. Secondly, support

Table 4.3: *Document classification on 3 classes with SpectrumBoost algorithm. Features are calculated both on a stemmed and non-stemmed version of the dataset.* Feature Extraction *column indicates which is the length of the sub-string used to build the features (e.g.* pN *refers to sub-strings of length N). Kernel matrix is computed with Nystrom approximation of order 5000 (which means that 5000 variables are extracted from each document). All the models are tuned with a Bayesian random search;* Training Time *refers to the retrain with the best set of hyperparameters.*

Algorithm	Feature Extraction	Time Feature Extraction	f1	AUC	Balanced Accuracy	Accuracy	Training Time
WITH STEMMING							
`XGBOOST`	p3	01:04:35	0.724	0.861	0.709	0.726	00:15:16
`XGBOOST`	p4	01:05:10	0.713	0.860	0.708	0.726	00:14:28
`XGBOOST`	p5	01:03:42	0.725	0.860	0.709	0.727	00:14:47
`XGBOOST`	p6	01:02:03	0.720	0.857	0.704	0.723	00:14:41
`XGBOOST`	p7	01:03:10	0.714	0.853	0.698	0.717	00:14:57
WITHOUT STEMMING							
`XGBOOST`	p3	00:56:43	0.724	0.860	0.709	0.726	00:16:23
`XGBOOST`	p4	01:00:24	0.727	0.861	0.711	0.729	00:12:52
`XGBOOST`	p5	01:03:33	0.727	0.861	0.710	0.729	00:14:33
`XGBOOST`	p6	01:05:09	0.723	0.858	0.707	0.725	00:15:32
`XGBOOST`	p7	01:06:02	0.723	0.856	0.707	0.725	01:04:53
`XGBOOST`	p3+p4+p5*		0.7196	0.8557	0.7041	0.7215	00:18:11
`XGBOOST`	1000 most important†		0.7251	0.8600	0.7096	0.7268	00:10:02

* Features extracted from spectrum $p = 3, 4, 5$ are summed element wise.

† 1000 most important features (renked by `XGBoost`) are extracted from each spectrum $p = 3, 4, 5, 6, 7$ and use as input variables for the model.

Table 4.4: *Document classification on 3 classes using BERT variants.* Runs *column indicates the number of tests done for each model, and the scores represent the mean values.* Training Time *refers to the time needed to fine tune the model.*

Algorithm	Runs	f1	AUC	Balanced Accuracy	Accuracy	Training Time
WITH STEMMING						
dbmdz/bert-base-italian-xxl-uncased*	2	0.718	0.836	0.708	0.721	03:00:02
dbmdz/bert-base-italian-xxl-uncased*	3	0.721	0.847	0.714	0.717	02:33:02
dbmdz/electra-base-italian-xxl-cased-discriminator*	2	0.716	0.842	0.707	0.718	06:30:00
dbmdz/electra-base-italian-xxl-cased-discriminator*	3	0.716	0.847	0.706	0.718	07:21:02
m-polignano-uniba/bert_uncased_L-12_H-768_A-12_italian_alb3rt0†	2	0.716	0.826	0.703	0.718	07:28:00
m-polignano-uniba/bert_uncased_L-12_H-768_A-12_italian_alb3rt0†	3	0.717	0.838	0.706	0.718	04:44:00
idb-ita/gilberto-uncased-from-camembert‡	2	0.718	0.848	0.706	0.721	05:08:03
idb-ita/gilberto-uncased-from-camembert‡	3	0.715	0.836	0.705	0.716	06:53:23
Musixmatch/umberto-commoncrawl-cased-v1††	2	0.715	0.839	0.709	0.715	03:16:14
Musixmatch/umberto-commoncrawl-cased-v1††	3	0.710	0.843	0.701	0.709	05:39:02
WITHOUT STEMMING						
dbmdz/electra-base-italian-xxl-cased-discriminator*	2	0.720	0.850	0.705	0.721	05:23:31
dbmdz/electra-base-italian-xxl-cased-discriminator*	3	0.720	0.848	0.722	0.709	05:55:28
dbmdz/bert-base-italian-xxl-uncased*	2	0.721	0.839	0.714	0.721	02:16:57
dbmdz/bert-base-italian-xxl-uncased*	3	0.720	0.848	0.715	0.718	01:55:49
m-polignano-uniba/bert_uncased_L-12_H-768_A-12_italian_alb3rt0†	2	0.716	0.833	0.713	0.717	02:17:52
m-polignano-uniba/bert_uncased_L-12_H-768_A-12_italian_alb3rt0†	3	0.717	0.835	0.714	0.717	02:28:23
idb-ita/gilberto-uncased-from-camembert‡	2	0.722	0.841	0.714	0.721	05:40:51
idb-ita/gilberto-uncased-from-camembert‡	3	0.717	0.840	0.711	0.715	04:12:02
Musixmatch/umberto-commoncrawl-cased-v1††	2	0.717	0.836	0.717	0.707	07:12:04
Musixmatch/umberto-commoncrawl-cased-v1††	3	0.715	0.839	0.714	0.714	03:08:18

* Model pretrained on Wikipedia dumps, text from Opus corpora and OSCAR corpus [33].
† Italian BERT trained over 200M twitters [37].
‡ Italian language model based on RoBERTa and camemBERT tokenization, trained on the OSCAR corpus [38].
†† Model architecture based on RoBERTa and trained using *SentencePiece* and *Whole Word Masking* [39].

tickets are not extensive pieces of text where long-term correlations between words are important. Instead, the meaning is enclosed inside few keywords, that may be repeated. In addition to that, spelling errors and typos are common. For these reasons, high value is assigned to sub-string with short length rather than to long sentences. As an example, the word "temperature" may appear as *temperatura*, *temp*, *temperat*, *temperature*, but the underlying meaning does not change. In such context, spectrum kernel is able to extract significant features, since it focuses on the repetition of small sub-strings.

Beyond performance metrics, considerations need to take into account also time and resources required by the algorithms. Calculation of spectrum features typically takes about one hour using a GPU and applying Nystrom approximation is required due to memory limit (order 5000 is the maximum achievable by the available machine that has 110 GB of RAM); XGBoost model training is quite fast on GPU (around 15 minutes) and occupies about 6 GB of memory. This makes it possible to perform parameter tuning in a reasonable amount of time and resources. On the other hand, BERT models need demanding processing resources (such as multiple very performant GPUs) since weights and architecture alone are very expensive in terms of hardware resources needed to use them. Moreover, training BERT models, even having access to large amounts of computing resources, requires a long time: the range shifts from 2 hours up to 7 (see Tab. 4.4), depending on the specific model variation. The consequence is the difficulty to perform parameters tuning.

In conclusion, the best performance is achieved by the spectrum kernel with $p = 4$ (which means that features are extracted by looking at sub-strings of length 4) in association with the XGBoost classifier. Its results can serve multiple scopes. For example, an automatic classification of maintenance tickets (*Assistance calls* dataset) can be implemented. In this way, the company can already identify the type of failure from the ticket and then send the appropriate technician. The result is a shorter downtime due to the failure. Secondly, tickets classified by SpectrumBoost can be used to construct a different dataset for the analysis performed in Chapter 3, which can lead to better performances if it mitigates the class imbalance problem.

5 Conclusions and Future Developments

This work focused on the application of Machine Learning in industrial maintenance. The particular use case was related to the development of those models to aspects connected to refrigeration systems starting from the available data, which consisted of alarms, maintenance operations logs, and maintenance tickets. Different methods were developed for the analysis and at the possible predictions of the failures, based on the combined study of different parts of the dataset.

In this Chapter, the results obtained are discussed in light of the entire set of researches described in this work; possible future developments are also proposed to extend and improve the investigated solutions.

Eventually, an additional piece of work is presented. It is less related to the model development aspect, but rather it focuses on the integration of the necessary computing pipeline based on the Amazon Web Services infrastructure.

5.1 Failure Prediction

In order to define a predictive model for the maintenance operations of the studied use case, the goal was first set to build an algorithm able to detect whether a failure is likely to happen the next day given the alarm history of the previous three days. The analysis is performed over a subset of the data, extracted following company indications, which specify 80 facilities and some keywords to filter failure reports in order to choose the ones related to faults in the cold chain. This results in a highly unbalanced sample: class 0, which indicates no faults, gathers 99.5% of the examples, while label 1 is assigned only to the 0.5% remaining. This disparity heavily influences the learning of the models, since they struggle to learn the minority class due to the lack of samples.

Four different solutions are proposed. XGBoost model displays the best performances, even if it is not specifically designed for time series. It describes every time step as an independent variable, such that raw alarm counts provide more information than their escalation. With the best hyperparameters configuration, obtained following a dedicated grid-search, the model reaches an AUC score of 0.756. Since the model outputs the probability of belonging to a certain label, the class is assigned by selecting a threshold as discussed in Sec. 3.2. As it is more important to detect failures, while it can be acceptable to have a non-negligible rate of false positives (mistake the correct operation as faulty), the suggestion is to set this threshold below 0.3, as shown by Fig. 3.2. At the probability threshold $p = 0.3$, almost 70% of failures are detected, and also the majority class is correctly recognized 65% of the time. If the setting requires a more conservative approach, the value can be lowered: at $p = 0.1$, nearly all the malfunction events are individuated, but the 0 class is mistaken half of the time.

NN-based solutions have been explored by using LSTMs, recurrent NNs specifically devoted to model time series. It has however been observed that none of the NN-based solutions tested was able to outperform the XGBoost approach. The LSTM network seems not to be able to understand the

data, and behaves like a random classifier, as stated by the AUC score below 0.5. This is a bit unexpected since this kind of network implements an architecture specific for data structures like time series. Probably, the problem resides in the sparsity of the data and in the fact that failures are not anticipated by an escalation in the alarms, but rather are sudden events. In this setting, long correlations in time may not be important, but instead, they trick the model and lead to worse results. In addition, they result to be less robust to the imbalance in the dataset with respect to XGBoost.

An alternative extension to the time series prediction based on LSTM can come from NLP-like approaches, which treats the sequence of alarms as a text, and learn their "language". However, similarly to the LSTM approach, the NLP-based algorithm shows poor performances, with an AUC score of 0.576; in addition, confusion matrices in Fig. 3.5 indicate that all the samples are assigned to similar probability values around 0.6, which prevent a clear separation between the two classes.

Fig. 3.6 suggests that XGBoost reaches a good classification accuracy for both classes (above 70% at $p = 0.3$) if it is not fed with the history of the alarms, but rather with the counts of the same day. These improvements may be due to the fact that a smaller number of variables is easier to learn and reduce the risk of overfitting. Hence, the problem is shifted to the prediction of the alarms for such a day given the three-day past. An LSTM network is applied for this purpose. Thus, the final ensemble model uses the recurrent network to forecast the time series to the next day, and then XGBoost to detect whether a failure happens or not. However, the overall LSTM + XGBoost pipeline shows an AUC of 0.66, so the combination of the two algorithms does degrade the performances. In addition, the ability to correctly classify both classes at the same time also vanishes. This result is not expected, since the LSTM shows an MAE metric of 0.032, so its predictions should be extremely similar to the real values.

The ensemble solution should be investigated more in the future. In particular, the LSTM section needs more attention, since it seems to be the cause of the low performances. Moreover, the ability to predict the number of alarms the successive days can be useful for other studies. For example, it should be possible to construct confidence intervals and try to detect failure with that base. Or, if the company provides more documentation about the critical values for a failure to be dangerous, it would be possible to compare the LSTM values to those ranges. Even combining this information with the XGBoost output could be a solution to obtain a more robust model.

Since the difficulty in the learning procedure is determined by the high imbalance in the data, future researches should make use of the results of the SpectrumBoost algorithm for the ticket classification to extract a more meaningful subset of data. Then all the proposed solutions may be investigated again, to see how much a different dataset influences the results.

5.2 Ticket Analysis

As an attempt to get an improvement over the results obtained with the Failure Prediction approach solely based on the *Alarm records* dataset, unsupervised learning approaches based on Natural Language Processing over the *Assistance calls* have been devised, expecting to be able to extract useful information from the maintenance requests. However, implementations of Latent Dirichlet Allocation methods weren't able to provide clear identification of topics belonging to different semantic areas. Fig. 4.6, in fact, clearly shows that the most relevant words for each cluster are not unique, but instead are shared across all the groups. In addition, the main topic cannot be easily extracted from the results, since LDA lacks interpretability. Alternative approaches based on a Doc2Vec model, with ad hoc training for this specific use case, have been explored. Also in this case however it has been observed that the model was not able to build a good latent features representation for the documents (see Fig. 4.7); moreover, the clusters extracted from this method once more showed strong overlaps of the most frequent words, not allowing to associate each group to a unique topic (Tab. 4.1). To further extend this approach, a pre-trained BERT model was thus used, as it represents the state of the art for NLP. This model was observed performing better in mapping each document into the embedded space: in Fig. 4.8 samples belonging to the same cluster are represented by contiguous points, which

do not happen with a Doc2Vec model trained from scratch. Nonetheless, the problem of interpreting the clusters' main topics persists (as shown by the top words in Tab. 4.2), which makes all of the studied models and methods not really suitable for extracting the relevant documents associated with faults in the cold chain.

The study of the *Operations* dataset allows reducing the unsupervised task to a supervised one making use of the technicians' annotations that associate each assistance ticket to a fault type. Two main approaches are tested: the proposed SpectrumBoost algorithm extracts features through the spectrum kernel and performs the classification with an XGBoost model; the second procedure involves fine-tuning of BERT models. Results in Tab. 4.3 and 4.4 quite surprisingly/interestingly indicate that the first technique delivers better results, with a gain of about 1% in both f1 and AUC scores compare to BERT. This outcome is interpreted as related to the nature of the document analyzed. The maintenance tickets do not typically feature long texts, but rather the focus is set to few core words which identify the nature of an issue. In this context, local features such as the ones extracted from the spectrum kernel play a more relevant role with respect to long-term dependencies and correlations between words that are unfolded by BERT.

Some proposals to improve performances of SpectrumBoost can be explored for future developments. The first one is to use the full kernel, without approximations. However, this approach requires more hardware resources, and may not be affordable to implement. Secondly, the combination of spectrum features obtained at different ranks can lead to a boost. The idea is to combine kernels with the length of sub-sequence $p \in [3, 7]$ using a 1-dimensional convolution: in this way, contributions of sub-strings with different lengths are joined together, and it can help to emphasize local word structure found at different levels. Another method is to join features from both spectrum and BERT latent space. The purpose is to merge local and long-range interactions in a single model.

The SpectrumBoost algorithm can serve two objectives. The first one is to automatically classify entries in *Alarms calls*. This solution can provide the company an insight into which issue caused a failure from the help request received. The consequence is that technicians specialized in that field may be immediately sent to repair the fault, instead of doing inspections to understand the problem. Both the inefficiency time and the costs can be reduced in this way. More failure types can also be included in the model, in addition to the three already considered. However, the class imbalance should always be taken into account. If the are not enough examples, adding categories may just result in performance degradation. A more robust proposal will be to include only the groups with a certain representation and include all the other cases into a single cluster. This solution is expected to allow to identify with high degrees of confidence the most statistically relevant faults, while still taking into account all the possible scenarios.

The second SpectrumBoost application is to predict the fault type in *Alarms calls*, and then use this information to extract requests connected to cold chain failures. Then, a more informative set of examples may be identified in order to train the failure prediction model.

5.3 Automated pipeline in AWS infrastructure

Amazon Web Services (AWS) provides an on-demand cloud computing platform that is composed of a variety of abstract infrastructure and computing building blocks. Their offer accommodates more than 200 different services and includes a mixture of infrastructure as a service (IaaS), platform as a service (PaaS), and packaged software as a service (SaaS). For a company, this service supplies compute, storage, network, and database resources needed for any project. The environment can be reconfigured easily, updated quickly, scaled up or down automatically to meet usage patterns and optimize spending cost, with a pay-as-you-go billing configuration.

In order to be implemented in the daily workflow of a big company, the Predictive Maintenance pipeline needs to be executed in such web service infrastructure. Fig. 5.1 shows a schematic view of a possible workflow. The main services implemented are [40]:

CloudWatch CloudWatch Events (or EventBridge) delivers a stream of real-time data from your own applications, Software-as-a-Service applications, and AWS services and routes that data to targets. It allows building event-driven architectures, which are loosely coupled and distributed. An event indicates a change in an environment or can be set as a scheduled occurrence. If it matches a configured rule (which specifies both events and targets), a signal is sent to the target.

Lambda Lambda allows to run code without provisioning or managing services and scales up automatically. It can be implemented in a variety of programming languages, such as Python through the `Boto3` package. Common usage cases comprehend running code in response to events or triggering AWS services.

Athena Amazon Athena is an interactive query service that makes it easy to analyze data directly in Amazon Simple Storage Service (Amazon S3) using standard SQL. It is a serverless service, meaning that there is no infrastructure to set up; it automatically scales up and allows to perform queries in parallel.

Simple Storage Service (S3) Amazon S3 is an object storage service that offers industry-leading scalability, data availability, security, and performance. Objects are contained inside buckets; they allow to organize the Amazon S3 namespace, identify the account responsible for data storage and transfer and allow to manage access control. The objects are the entities stored in S3 and consist of a set of data (opaque to S3) and metadata, which identifies it with sets of key-value pairs. Every object in Amazon S3 can be uniquely addressed through the combination of the web service endpoint, bucket name, key, and optionally, a version.

Glue AWS Glue is an ETL (Extract, Transform and Load) serverless service. It is built on top of an Apache Spark environment and it is designed to organize, cleanse, validate, and format data for storage in a data warehouse or data lake.

Step Function Step Functions is a serverless orchestration service that lets you combine AWS Lambda functions and other AWS services. It is based on a state machine, which is a workflow made up of tasks, which represents a single unit of work carried out by other AWS services. Possible use cases include function orchestration (running a set of Lambda functions where each one receives the output of the previous), error handling and notification, and process parallelization.

SageMaker Amazon SageMaker is a machine learning service with algorithms optimized to run efficiently against extremely large data in a distributed environment. It offers flexible distributed training options to run your own models.

Forecast Amazon Forecast is a fully managed service that uses statistical and machine learning algorithms to deliver highly accurate time-series forecasts. It automates the ML tasks (model selection, tuning) and implements state-of-the-art solutions.

The pipeline in Fig. 5.1 involves the following steps:

- An event such as putting new data in an S3 bucket or a scheduled one activates the CloudWatch service which in turn triggers a Lambda function.
- The Lambda starts an Athena query over the database which results are stored inside an Amazon S3 storage in `csv` or `hd5f` format.
- Elements added to S3 automatically trigger a Lambda function which may perform some simple task (such as cleaning or preparing buckets) and feeds the results of the query to the Glue job.
- Glue performs the data preparation and preprocessing. It allows to handle a huge amount of data and transform them in a distributed way. Cleaned and transformed data are then rewritten into an S3.
- Two different streams can be implemented to perform the analysis. If time-series predictions are needed the goal is Forecast, while SageMaker is to choose if a custom ML model is requested. Independently of the service, an AWS Step Function workflow handles the different steps required

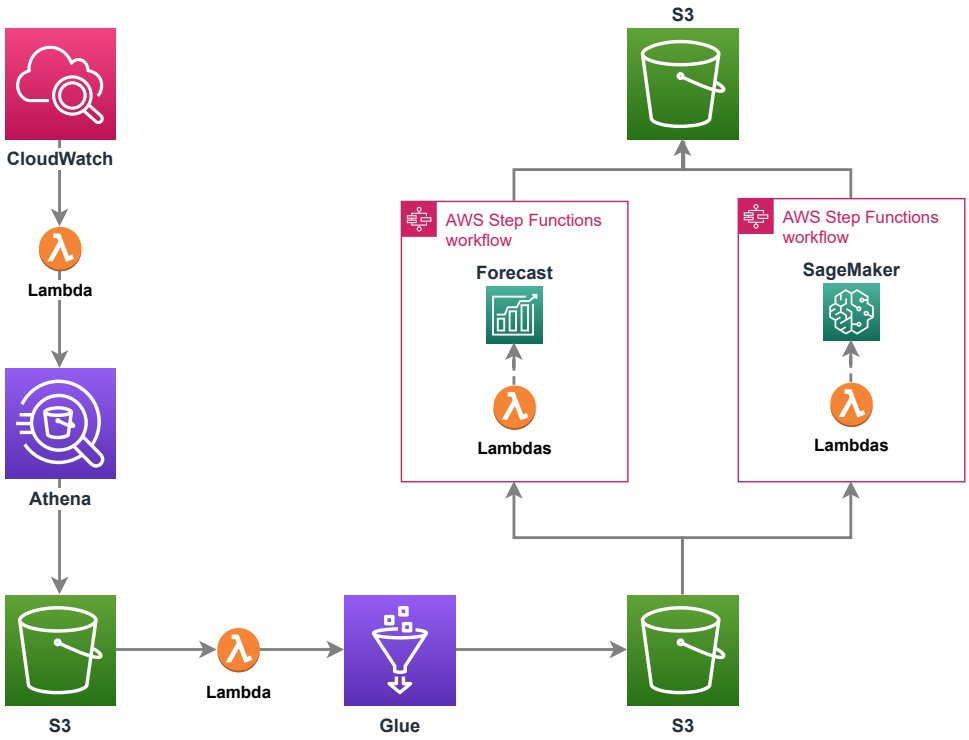


Figure 5.1: *Schematic representation of the AWS pipeline to train and apply the predictive maintenance algorithm using cloud services.*

for the study. In particular, it comprehends Lambda functions that merge training and model selection steps, and routines to make the actual predictions. In addition, since in a daily scenario the flux and amount of data are huge, monitoring tools and error notification and handling must be implemented in the pipeline.

- Eventually results are stored in a new S3 storage and can be further accessed by other services to display results in appropriate dashboards.

An infrastructure of this kind is necessary for an industrial context to make in action the algorithms developed in this study. It allows automating the whole procedure, without requiring any human intervention. Moreover, results can be extracted, further processed, and distributed among all the facilities using the same environment. Future improvements and models could also be developed and included using this flexible suite, without the need to rebuild the infrastructure from scratch.

Bibliography

[1] T. Carvalho, F. Soares, R. Vita, R. Francisco, J. Basto, and S. G. Soares Alcalá, "A systematic literature review of machine learning methods applied to predictive maintenance," *Computers & Industrial Engineering*, vol. 137, p. 106024, 09 2019.

[2] G. A. Susto, A. Schirru, S. Pampuri, S. McLoone, and A. Beghi, "Machine learning for predictive maintenance: A multiple classifier approach," *IEEE Transactions on Industrial Informatics*, vol. 11, no. 3, pp. 812–820, 2015.

[3] Dask Development Team, *Dask: Library for dynamic task scheduling*, 2016. [Online]. Available: https://dask.org

[4] P. Mehta, M. Bukov, C.-H. Wang, A. G. Day, C. Richardson, C. K. Fisher, and D. J. Schwab, "A high-bias, low-variance introduction to machine learning for physicists," *Physics Reports*, vol. 810, p. 1–124, May 2019. [Online]. Available: http://dx.doi.org/10.1016/j.physrep.2019.03.001

[5] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization." *Journal of machine learning research*, vol. 13, no. 2, 2012.

[6] E. Brochu, V. M. Cora, and N. de Freitas, "A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning," 2010.

[7] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica, "Tune: A research platform for distributed model selection and training," *arXiv preprint arXiv:1807.05118*, 2018.

[8] T. Head, M. Kumar, H. Nahrstaedt, G. Louppe, and I. Shcherbatyi, "scikit-optimize/scikit-optimize," Sep. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.4014775

[9] Tensorflow. Classification on imbalanced data. [Online]. Available: https://www.tensorflow.org/tutorials/structured_data/imbalanced_data

[10] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: Synthetic minority over-sampling technique," *J. Artif. Int. Res.*, vol. 16, no. 1, p. 321–357, Jun. 2002.

[11] G. Lemaître, F. Nogueira, and C. K. Aridas, "Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning," *Journal of Machine Learning Research*, vol. 18, no. 17, pp. 1–5, 2017. [Online]. Available: http://jmlr.org/papers/v18/16-365.html

[12] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: ACM, 2016, pp. 785–794. [Online]. Available: http://doi.acm.org/10.1145/2939672.2939785

[13] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[14] D. E. Rumelhart and D. Zipser, "Feature discovery by competitive learning," *Cognitive Science*, vol. 9, no. 1, pp. 75–112, 1985. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0364021385800100

[15] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola, *Dive into Deep Learning*, 2020, https://d2l.ai.

[16] C. Olah. Understanding lstm networks. [Online]. Available: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

[17] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *the Journal of machine Learning research*, vol. 3, pp. 993–1022, 2003.

[18] M. D. Hoffman, D. M. Blei, C. Wang, and J. Paisley, "Stochastic variational inference," *Journal of Machine Learning Research*, vol. 14, no. 4, pp. 1303–1347, 2013. [Online]. Available: http://jmlr.org/papers/v14/hoffman13a.html

[19] D. MacKay, D. Kay, and C. U. Press, *Information Theory, Inference and Learning Algorithms*. Cambridge University Press, 2003. [Online]. Available: https://books.google.it/books?id=AKuMj4PN_EMC

[20] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013.

[21] Q. V. Le and T. Mikolov, "Distributed representations of sentences and documents," 2014.

[22] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2019.

[23] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," 2019.

[24] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter," 2020.

[25] K. Clark, M.-T. Luong, Q. V. Le, and C. D. Manning, "Electra: Pre-training text encoders as discriminators rather than generators," 2020.

[26] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2017.

[27] I. Lauriola, S. Campese, A. Lavelli, F. Rinaldi, and F. Aiolli, "Exploring the feature space of character-level embeddings."

[28] D. Arthur and S. Vassilvitskii, "K-means++: The advantages of careful seeding," in *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '07. USA: Society for Industrial and Applied Mathematics, 2007, p. 1027–1035.

[29] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[30] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017.

[31] R. Řehůřek and P. Sojka, "Software Framework for Topic Modelling with Large Corpora," in *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. Valletta, Malta: ELRA, May 2010, pp. 45–50, http://is.muni.cz/publication/884893/en.

[32] C. Sievert and K. Shirley, "LDAvis: A method for visualizing and interpreting topics," in *Proceedings of the Workshop on Interactive Language Learning, Visualization, and Interfaces*. Baltimore, Maryland, USA: Association for Computational Linguistics, Jun. 2014, pp. 63–70. [Online]. Available: https://www.aclweb.org/anthology/W14-3110

[33] S. Schweter, "Italian bert and electra models," Nov. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.4263142

[34] J. Tiedemann, "Parallel data, tools and interfaces in opus," in *Proceedings of the Eight International Conference on Language Resources and Evaluation (LREC'12)*, N. C. C. Chair), K. Choukri, T. Declerck, M. U. Dogan, B. Maegaard, J. Mariani, J. Odijk, and S. Piperidis, Eds. Istanbul, Turkey: European Language Resources Association (ELRA), may 2012.

[35] P. J. Ortiz Suárez, L. Romary, and B. Sagot, "A monolingual approach to contextualized word embeddings for mid-resource languages," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Online: Association for Computational Linguistics, Jul. 2020, pp. 1703–1714. [Online]. Available: https://www.aclweb.org/anthology/2020.acl-main.156

[36] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Online: Association for Computational Linguistics, Oct. 2020, pp. 38–45. [Online]. Available: https://www.aclweb.org/anthology/2020.emnlp-demos.6

[37] M. Polignano, P. Basile, M. de Gemmis, G. Semeraro, and V. Basile, "AlBERTo: Italian BERT Language Understanding Model for NLP Challenging Tasks Based on Tweets," in *Proceedings of the Sixth Italian Conference on Computational Linguistics (CLiC-it 2019)*, vol. 2481. CEUR, 2019. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85074851349&partnerID=40&md5=7abed946e06f76b3825ae5e294ffac14

[38] G. Ravasio and L. D. Perna, "Gilberto: An italian pretrained language model based on roberta." [Online]. Available: https://github.com/idb-ita/GilBERTo

[39] L. Parisi, S. Francia, and P. Magnani, "Umberto: an italian language model trained with whole word masking." [Online]. Available: https://github.com/musixmatchresearch/umberto

[40] Amazon Web Services. Aws documentation. [Online]. Available: https://docs.aws.amazon.com/index.html