UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO
DI INGEGNERIA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA MAGISTRALE IN
ICT FOR INTERNET AND MULTIMEDIA

TITOLO

AI-Driven Technical Report Analysis and Failure Prediction: A Case Study in Intelligent Maintenance System

Relatore: Prof. Simone Milani
Tutor: Davide Bagnoli

Laureando: Leonardo Brunetti

ANNO ACCADEMICO 2024 –2025

Data di laurea 08/07/2025

Abstract

This thesis presents the development of an automated system for analyzing test documents (Device History Records, or DHRs) and corresponding maintenance reports, with the goal of identifying possible correlations between the initial configuration of electronic devices and failures observed during their lifecycle. The project is structured in two main phases: the first involves extraction of raw text data and transformation into a structured format; the second focuses on training a machine learning model to identify patterns that may suggest relationships between specific test parameters and subsequent technical issues. The work explores the details of the test procedures performed on the devices, including key parameters such as power levels, energy output, and pulse duration. It also faces challenges about working with semi-structured documents, especially when dealing with handwritten values and different layouts. Special attention is given to the problem of connecting test data to service reports when unique identifiers are missing or incomplete. The results show the potential of using historical data to support predictive maintenance, showing that a data-driven approach can improve fault detection and support decision-making in industrial settings. The system also sets up the ground for future improvements in document automation and quality monitoring processes.

Table of contents

Introduction **1**

Background and motivation 1

State of the art 2

Main objectives 3

Overview of the project workflow 4

Summary of key activities 4

Chapter 1: the Role of AI in Technical Report Analysis **6**

1.1 Technical Reports in Maintenance Engineering 6

1.2 Challenges in Analyzing Technical Reports 8

1.3 Processing Pipeline: from PDF to Structured Data 9

1.4 AI Models and Prompt Engineering for Document Understanding 13

1.5 Overcoming Common Challenges in Technical Report Analysis 16

Chapter 2 – Data Preparation and Extraction for Predictive Analysis **17**

2.1 Data Collection from Technical Reports 17

2.2 Features Description 18

2.3 Cleaning and Preprocessing Raw Data 19

2.4 Understanding Feature Relevance for Predictive Models 22

2.4.1 Correlation matrix 23

2.4.2 Feature Selection and Point-Biserial Correlation 25

2.4.3 Visual Comparison Between Classes 27

2.5 Importance of Data Quality in Predictive Analysis 31

Chapter 3 – Developing and Training the AI Model for Failure Prediction **33**

3.1 Overview of the Failure Prediction Model 33

3.2 Selection of the Appropriate AI Model 35

3.3 Training the Model with Historical Data 40
3.4 Fine-Tuning and Evaluating Model Performance 43
3.5 Integrating the Model into a Maintenance System 48
Chapter 4 – System Evaluation 51
4.1 Testing the AI Model: Evaluation Metrics and Criteria 51
4.2 Validation and Test Results 52
4.2.1 Class-wise Performance Analysis 53
4.2.2 Comparison with Training Phase 55
4.2.3 Metric's Interpretation 55
4.2.4 Implications for Deployment and Future Work 56
4.3 Feature Importance and Explainability 56
4.4 Limitations and Edge Cases 58
4.5 Next Steps and Future Improvements 60
Conclusion 62
Appendix A – Code Structure 64
Appendix B – Prompt Examples 66
B.1 Classification Prompt 66
B.2 Extraction Prompt with JSON Template 67
B.3 Prompt Execution Logic 68
B.4 Why Prompt Engineering Matters 68
Bibliography 69
Web References 70

Introduction

Background and motivation

In recent years, AI has moved very quickly from being a research topic to become a practical tool used across many industries. Sectors like manufacturing, logistics, healthcare, and energy have adopted AI to improve efficiency, reduce human error, and gain insights that would otherwise be time-consuming and difficult to process manually. Whether used for optimization, automation, demand forecasting, or whatever function, AI helps make operations faster and smarter.

One of the sectors where this transition is particularly visible is industrial maintenance. Traditionally, maintenance has followed either a reactive or preventive approach, fixing things after they break, or scheduling maintenance at regular intervals. Both methods can be inefficient: the first might lead to unexpected downtimes, the second often results in unnecessary interventions. Predictive maintenance aims to solve this, and it heavily relies on historical data. One of the main sources of such data is technical documentation, including Device History Records (DHRs), test reports, service logs, and incident summaries.

DHRs are structured technical documents that track the full lifecycle of an industrial product from initial testing and calibration to maintenance. They are typically produced during manufacturing and include detailed measurements, configurations, and quality control notes. In many industries, they are mandatory, as they serve as proof of compliance with safety and reliability standards. Their completeness and traceability make them essential for diagnosing failures, identifying recurring issues, and improving future production. These documents contain a lot of valuable information about machine behavior and configuration. However, they are often reviewed manually, slowing the process and makes it prone to errors. That's why interest in AI keeps growing: techniques like NLP (Natural Language Processing) and ML (Machine Learning) can help automate the analysis of technical reports.

The core idea of this project is to turn unstructured or semi-structured documents into structured datasets, where fields like test parameters, configuration values, or defect notes are automatically extracted and exposed in a precise format. Once structured, this information can

be connected to respective outcomes to find recurring patterns and risk factors. If successful, this approach helps reduce the time spent on documentation and brings out insights that might otherwise go unnoticed.

This project explores how AI can support predictive maintenance by automatically extracting and analyzing data from technical reports, with the goal of improving efficiency and quality of the process. The devices that had been analyzed in this study are high-precision electronic instruments that go under strict testing before and after deployment. These tests typically include laser calibration, energy output measurement, and galvo alignment procedures. The exact product line is under NDA, but the data reflects the kind of structured and semi-structured technical validation used in industrial systems.

The project was developed within the context of my internship at **Computer Gross**, a leading technology distributor in Italy. I was part of a broader team effort focused on improving internal processes related to document automation and maintenance data analysis. The system was designed for a real client in the high-tech manufacturing sector and developed in collaboration with a business partner. I was involved through the Cloud & AI team, where I specifically follow the watsonx platform, IBM's AI environment.

My contribution focused on the design and implementation of the AI components, starting from document understanding. I configured the *IBM watsonx.ai* platform, selected the most suitable vision-language model (Pixtral) for field extraction, wrote and deployed prompt templates, and validated the output. On the predictive side, I worked on dataset preparation, model training with Random Forest, and integration of explainability tools using SHAP. The rest of the pipeline, particularly the orchestration logic and document workflow, was developed with other team members specialized in document processing.

State of the art

Technical documentation such as Device History Records (DHRs) and service reports are very important in the maintenance process of industrial systems. Historically, the analysis of these documents relied on manual inspection or, at best, on rule-based systems using keyword matching, regular expressions, or fixed OCR templates. These approaches, yet relatively simple to implement, are also extremely fragile: they often fail when documents change even slightly

in format, when data is handwritten, or, in general, when the structure is inconsistent across records. To overcome these limitations, the last few years have seen the emergence of more advanced document AI systems, which include tools such as Amazon Textract, Google Document AI, and Microsoft Form Recognizer, which combine OCR with layout detection and some basic entity extraction, these systems are still constrained by predefined templates or limited to well-structured forms. In parallel, academic research introduced the idea of large language models (LLMs) that are aware of the layouts that combine textual and visual information from documents. These models have shown excellent results in tasks like document classification, key-value extraction, and form parsing. However, they often require supervised fine-tuning on large datasets and are not that easy to adapt to new layouts without retraining. Recently, the introduction of vision-enabled large language models (LLMs) such as GPT-4V, Claude 3 with vision, and domain-specific tools like watsonx with visual capabilities has opened a new path. These models can work over both image and text, use prompt-based instructions, and generate structured outputs directly, even without retraining. This represents a big change, especially for semi-structured documents like DHRs, where layout and content vary significantly from one another and may also include handwritten annotations.

The system presented in this work builds on this new generation of tools, leveraging a visual LLM to classify, understand, and extract technical data from complex industrial documentation. Compared to previous methods, this solution does not require layout-specific configurations, nor annotated training data, and can be extended to new document types simply by writing new prompts. This combination of flexibility, precision, and low setup cost defines a new state of the art in document processing applied to maintenance engineering.

Main objectives

Based on the context described above, the aim of this thesis is to design and evaluate an AI-based system capable of extracting useful information from technical documentation, to use it for predictive maintenance. In particular, the system processes DHRs and connects them to the related service reports, with the main goal of finding the most relevant parameters or initial configurations that may be linked to a malfunction. The main pipeline can be summarized as follows:

- Clean raw documents to remove unuseful or low-quality data.

- Extract useful features such as test parameters and operational settings.
- Connect this information to historical maintenance events and find correlations.
- Build a machine learning model that can predict the likelihood of failure based on initial configuration data.
- Validate the system's performance and identify possible improvements for future development.

Overview of the project workflow

The system is built as a multi-step pipeline, where in each stage a specific part of the process is handled.

It starts with a pre-processing phase, where the documents are checked for quality and filtered. Short or incomplete files are excluded, and the rest are segmented page by page. Each page is then analyzed using an AI model that includes the use of natural language understanding (NLU) and visual layout processing, namely OCR. This allows the system to recognize important fields and pass over the issues of handwritten form or variations from document to document. The extracted data is validated, structured, and stored in a database, ready for analysis.

In the final stage, the structured data is used to train a predictive model, which learns to associate test parameters with maintenance reports that will come later. The model is evaluated to measure how well it can identify patterns that lead to future failures.

Summary of key activities

Throughout the project, the following main activities were carried out:

- Data collection and filtering: Over 1300 DHR documents were reviewed, and a quality filter was applied to focus on those most useful for analysis.
- AI-driven document processing: A large language model with visual capabilities was used to extract handwritten and structured data from PDFs.
- Feature engineering and dataset building: the extracted data was cleaned and reduced to a final set of meaningful features.

- Model training and testing: a Random Forest classifier was trained to predict if a device would require service in the future, based on its original test values.
- Evaluation and insights: the system's performance was evaluated, and the most relevant features were identified, highlighting which parameters are most likely linked to device issues.

Chapter 1: the Role of AI in Technical Report Analysis

1.1 Technical Reports in Maintenance Engineering

In maintenance engineering, technical reports are essential tools to understand how machines behave over time and to document issues that come out during their lifecycle. These documents are usually compiled after field interventions and are meant to describe what was observed, the actions that were taken, and the results that followed. They provide a detailed and traceable history of a device's condition and performance. The devices that had been analyzed in this study are high-precision electronic instruments that go under strict testing before and after deployment. These tests typically include laser calibration, energy output measurement, and galvo alignment positioning.

The exact product line is under NDA, but the data reflects the kind of structured and semi-structured technical validation used in industrial systems.

There are different types of technical reports:

Device History Records (DHRs) are often used to track test results and configuration data before a machine is shipped. In practical terms, a DHR might contain a full page listing the results of laser alignment tests, with precise numeric values for pulse energy, duration, and calibration tolerances. They are usually filled out during factory testing and they are meant to reflect how the machine behaved in a controlled setting, before being shipped.

Service reports, on the other hand, are compiled after the device has been delivered and installed, usually following an on-site maintenance activity. They document the context of the intervention, the problem reported by the user (if any), the actions taken by the technician, and the final outcome. These reports are often much less structured than DHRs: they may include short notes, lists of replaced components, environmental observations, or even photos. Unlike the DHR, which aims to certify that the product passed a standard testing phase, the service

report tells the story of what went wrong in the field, and how it was fixed. It reflects the real-world usage of the device, often under suboptimal conditions.

While the DHR is standardized, consistent, and formatted for quality checks, the service report is informal, variable, and problem oriented. These differences are exactly what makes them hard to analyze together, and why AI is needed to resolve the issue and bridge the gap.

Even though each format has its own function, they all share the same challenge: how to capture useful information in a way that can be understood and reused. These documents are important, not just for compliance with regulations, but also for internal communication between maintenance teams, engineers, and quality control. A well-written report does more than describe an issue, as it helps others understand the reasoning behind decisions and identify patterns over time. Despite their value, though, technical reports are often archived without further use. They become static text, hard to access and analyze, and that's where AI tools can help bring structure to this data and turn it into something actionable.

Due to confidentiality constraints, complete DHRs cannot be published. However, it is possible to share representative visualizations or anonymized excerpts to illustrate the kind of information they contain. These simplified formats, often structured as JSON objects, are used throughout this project to describe how the system processes and interprets technical data. Figure 1.1 shows a human-readable tabular representation of a DHR segment originally formatted as JSON.

Step	1
Method	N/A
Test-Step definition	N/A
Limit values	N/A
Measured value	N/A
Test condition	N/A
Result	ok

Step	2
Method	N/A
Test-Step definition	N/A
Limit values	N/A
Measured value	N/A
Test condition	N/A
Result	ok

Step	3
Method	N/A
Test-Step definition	N/A
Limit values	N/A
Measured value	N/A
Test condition	N/A
Result	ok

Figure 1.1 – Example of structured representation of extracted DHR test data

1.2 Challenges in Analyzing Technical Reports

Technical reports hold a lot of useful information, but extracting it automatically is not that simple. The difficulty lies in the fact that different technicians might use different tones, abbreviations, or personal writing styles, and there's no strict format to follow: one might describe a problem in detail, another one can do it by summarizing it in just a few words, etc. And this variation makes it hard to extract structured data in a consistent way, because it would require having a tool able to find a structure and retrieve information independently from its shape. Adding to complexity, many reports include handwritten parts. Sometimes it's just a note or a number; in other cases, entire documents are filled out by hand and later scanned. This introduces issues in terms of both readability and structure, because aside from just reading messy handwriting, the problem is also about interpreting unstandardized data formats. Another issue lies in the limits of traditional automation tools. Rule-based or keyword-based systems often struggle when documents vary in terms of terminology or structure. These approaches depend on solid templates or rigid expectations. For example, if a label is missing or phrased differently, the extraction fails. In contrast, large language models (LLMs) can do reasoning and understand the context, and adapt to the circumstances, even in noisy or non-structured

documents. While not perfect, they allow for a more flexible and robust form of extraction that does not rely on predefined layouts or annotations. Moreover, the signal-to-noise ratio in the data was low. Some failure patterns were difficult to detect, nonlinear, or masked by variability. Even with cleaning and feature selection, many cases remained borderline and hard to classify confidently. In the end, the model had no information about the context or timeline of each device: it treated every device as an isolated case, without knowing anything about its usage history, production batch, or past events. For sure, these factors could have made the predictions easier to understand and more reliable.

These challenges really shaped and reshaped the design of the entire pipeline, from prompt-based extraction to model tuning. The system was built not just to extract and classify, but also to adapt to incomplete, noisy, and imbalanced data in a flexible way.

1.3 Processing Pipeline: from PDF to Structured Data

In this stage of the project, AI was used to study complex technical documents to find significant relationships between how a device was set up at the beginning and the problems it might have later.

This was not a typical natural language processing task, because in this case, instead of working with regular text, the AI had to deal with semi-structured data coming from data extracted from manufacturing and quality control steps. The documents (DHRs) include many technical values like power, energy, pulse duration, and readings from optical sensors. As mentioned before, these files had a starting structure of sheets to be compiled, but the service reports often had different formats and included handwritten numbers, which made them hard to read automatically. To handle this, the team used AI tools that combine language understanding and visual recognition.

The extraction pipeline was implemented through the **IBM watsonx.ai platform**, which provides access to a variety of large language models (LLMs) through a unified API. This flexibility allowed us to experiment with different model families and prompt configurations, adjusting parameters such as temperature, maximum tokens, and repetition penalty to better control the output. Among the models considered were **Granite**, IBM's internal series of decoder-only transformer models; **LLaMA 2**, a widely used open-source model; and **Pixtral**,

a vision-language model designed for multimodal tasks. Granite was initially considered for its robust performance on language-only benchmarks and its integration within IBM's governance ecosystem. However, this is not a multimodal model and lacks native support for image inputs, which makes it unsuitable for analyzing scanned documents containing handwritten sections or layout-sensitive content.

LLaMA 2 is powerful in general-purpose language generation but was inefficient in this context. In preliminary tests, it produced verbose and inconsistent outputs, even when using carefully constructed prompts with strict output format constraints. This verbosity was particularly problematic in our case, where the goal was to extract only structured fields in JSON format with no further explanation. For this reason, the final prompt was explicitly designed with repetition and formatting enforcement to discourage open-ended responses.

The selected model was **Pixtral-12B-2409**, a multimodal decoder model made available through watsonx.ai. Pixtral consists of a 12-billion-parameter text decoder paired with a 400-million-parameter vision encoder. It is trained using interleaved image and text data and supports variable image sizes and long input sequences (up to 128k tokens). It shows good performances across visual and text tasks, which makes it particularly well-suited for our use case. In this project, Pixtral was used to analyze scanned document pages and extract key fields based on prompt instructions, without requiring any supervised fine-tuning or annotated datasets.

For this purpose, the best solution was considered to be the Pixtral model. It was chosen because it can read handwritten text and convert it into JSON format, which is easier for computers to work with. Pixtral also worked well with different types of document layouts, so it did not need to be retrained each time the format changed.

The AI pipeline was designed to manage each step of the processing in a structured way:

- PDF documents were converted into images; the module *create_orders_pdf* (see appendix 1 for description module) also filtered documents with less than 25 pages since shorter ones are usually non-relevant as they might lack useful content. Each PDF was split into single pages to better control the next following steps.

- Each page image was classified using Pixtral as a language model, accessed through the watsonx platform as implemented in the *classify_orders_image.py* module. The model was prompted to extract key metadata from each page. The relevance of this step lays in the importance of organizing the processing and assigning the correct extraction actions based on the page's category.

- Next, each page image was classified using a language model integrated with computer vision (watsonx), as implemented in *classify_orders_image.py*. The model was prompted to extract key metadata from each page, such as the page code, revision number, and page number. This step was to make sure that the correct extraction logic is used based on the page's category.

- Now that pages are classified, they were passed to the state-machine-driven extraction system *(img_extraction_data)* that processed each image independently using a one-shot prompting approach. Hence, each page was analyzed with a different prompt tailored to its structure, and the model's output was a JSON format file that captured the data extracted from that page. As a result, this method is highly flexible even when handling non-standard or handwritten documents.

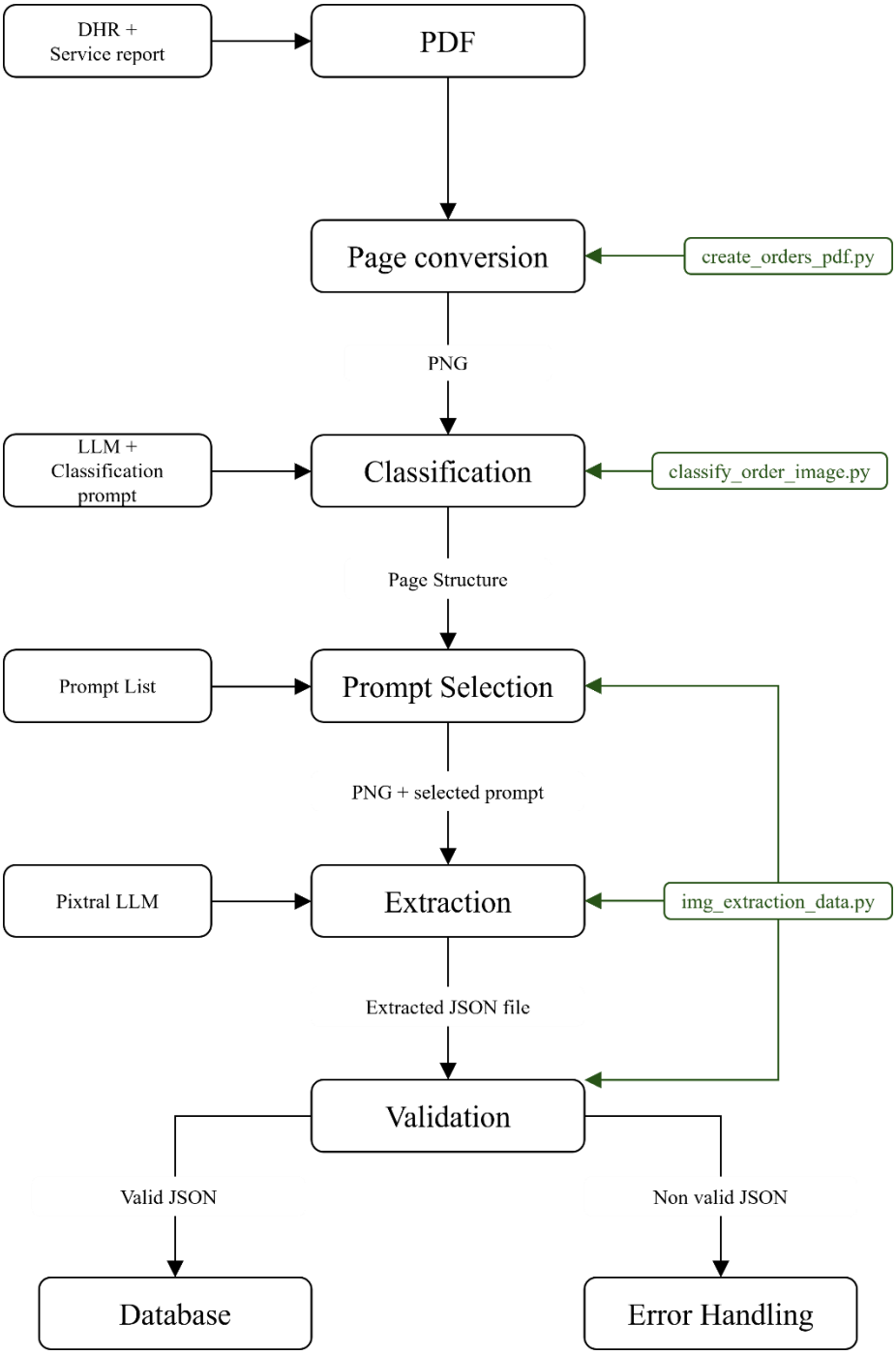


Figure 1.2 – Modular state diagram of the document extraction pipeline, with respective Python modules for each processing step.

All the pages of each document have been processed using this modular system that made it easier to lead the analysis step by step. Furthermore, in case of introduction of new categories,

this system allows to implement a new prompt for a new document category, because it only requires writing a new prompt and updating the classifier with the new page code.

1.4 AI Models and Prompt Engineering for Document Understanding

After defining how each page would be routed through the pipeline, the next focus was on how the content would actually be extracted. This required a system that could understand not only text but also content and layout. As explained in the previous section, the model used for extraction was accessed through the IBM watsonx API, and it supported both image and text inputs. To guide the model's behavior, very precise instruction was written using LangChain [LangChain, n.d.] (the main protagonist in handling dynamic prompts interacting with watsonx model): these instructions were written to clearly tell the model what to look for in the image and how to present the result. The prompt content was stored in some text files that were called after the pages were classified, and generally they force a strict output structure. In particular, the model was required to return data in a predefined JSON format, using "null" for missing values and avoiding any extra explanation text.

Each prompt was matched to a specific page type, identified earlier in the classification phase. The correct prompt was selected automatically based on the page's code, which had been extracted using a regular expression implemented in the function. This function scanned the model's classification output and returned a standardized string like [*SC_TBO000038, REV_B, 1]*, which was used to route the page to the right path. Thus, based on this classification, the right JSON template is copied into the prompt, substituting the correct line and served as a landmark for the extraction. The document is then processed following the instruction belonging to the detected category, so that the well-suited prompt is used for extraction information of that specific document. To give a reference, the category *SC_TBO_REV-1* means:

- SC_TBO = document category
- REV_B = revision tag
- -1 = page number

This system allowed the model to extract complex data from very different layouts without needing to be retrained. Adding a new document type was just a matter of writing a new prompt

and linking it to a page code. The output is always in JSON, and it was easy to validate and ready for analysis.

Overall, this combination of prompt engineering, regex-based classification, and structured output made the extraction process fast, flexible, and reliable. It would have been otherwise a manual and error-prone task. Down below are shown an example of a prompt template (for full reference, see Appendix B) and the example of the JSON file that is desired, used for one-shot encoding.

```
Your duty is to extract the information contained in the table found in the
image and export it into a json_format.

[. . .]

Blank data or null data should be "null". DON'T make up data.

[. . .]

Write only the json_format.
```

Figure 1.3 – Example of prompt template

```
{
  "table_data": {
    "Collaudo Ottico Intermedio": {
      "Posizionamento Galvo": {
        "L1": "int", // 5-digit code
        "L2": "int"  // 5-digit code
      },
      "Verifica potenza in uscita": {
        "calibration Laser 1": {
          "short pulse": {
            "power output (W)": "float", // 20-50
            "t (us)": "int"              // 500-700
          },
          "medium pulse": {
            "power output (W)": "float",
            "t (us)": "int"
          },

          [...]

        },
        "calibration Laser 2": {

          [...]

        }
      }
    }
  }
}
```

Figure 1.4 –Example of JSON to be generated.

This approach highlights the importance of custom prompt engineering in industrial applications. Unlike generic language tasks, technical extraction requires precise control over the model's behavior. The prompt becomes a central part of the system design because of its domain knowledge, definition of expected structure, and output constraints. In this case, prompt templates were not just a tool for formatting, but to link unstructured and structured data.

1.5 Overcoming Common Challenges in Technical Report Analysis

The irregularity and noise typical of technical documents were not treated as something to eliminate, but as part of the system's starting point. Rather than trying to force standardization, the solution was to create a process that could adapt to the real structure of the data, even when that structure was unreliable.

The system was built on a state-machine architecture, where every stage followed a clear sequence:

- Preprocessing
- Classification
- Prompt selection
- Data extraction
- Validation.

This allowed each step to be managed independently and gave better control when something went wrong. For example, if a page failed extraction, it could be reprocessed without having to restart the entire workflow. A second major decision was to keep the system modular: each component, like the one responsible for classifying documents or the one handling extraction, was developed separately. This way, adding support for a new document format did not require changing the full pipeline, only adjusting the prompt or classifier. This made the system more reusable and easier to maintain, both in this project and for future needs.

During testing, it also became clear that not all documents were useful. In fact, very short DHRs often lacked the technical detail needed for analysis. Including them would have wasted resources and added noise, so a simple pre-filter was added to exclude these cases automatically. In the end, these design choices made the system more stable and easier to expand. Even when full automation was not possible, the structure allowed for partial automation and fast recovery. This is how a set of challenges has been transformed into a scalable solution.

Chapter 2 – Data Preparation and Extraction for Predictive Analysis

After documents were processed and the information extracted using the right prompt, as described in the previous chapter, the next step was to prepare the data for predictive analysis. At this point, the challenge was finding the right way and tools to make sure the dataset was clean and useful from the perspective of a machine learning model, as the data would be used to train it. The quality and structure of the dataset have a direct impact on how well a model can learn from it, so this phase required aimed technical decisions with some caution about what to keep and what to discard. Most of the data came from DHRs, and included numerical values related to test parameters. This data was stored in tabular format, but the dataset still needed to be refined, in fact, some rows were incomplete, some features were empty or noisy and some devices were difficult to connect to the corresponding service report.

From here on, the work was to filter out and clean data, followed by merging different sources and defining a target variable (that would be a binary one: failure yes/no) to use for classification.

The chapter goes through these steps, it shows how raw data was transformed into a structured dataset ready for training, and it also focuses on the relevance of having reliable input when working with predictive systems, especially in industrial contexts where decisions based on bad data can lead to major mistakes.

2.1 Data Collection from Technical Reports

The starting point for building the final dataset was the output of the extraction process: a structured table where each row represents a device and includes the technical parameters recorded during its test phase. At this stage the values collected from DHR were available in tabular form, still not connected to real world outcomes. To enable predictive analysis, it was necessary to bring in a second source of information, which is the service reports explained in

the section above, that contain records of interventions and are stored in a separate dataset. The idea was to link the two sources (DHR test result and service records) so that each device would also have a label indicating whether or not it later required intervention. A simplified example of the mapping is shown below:

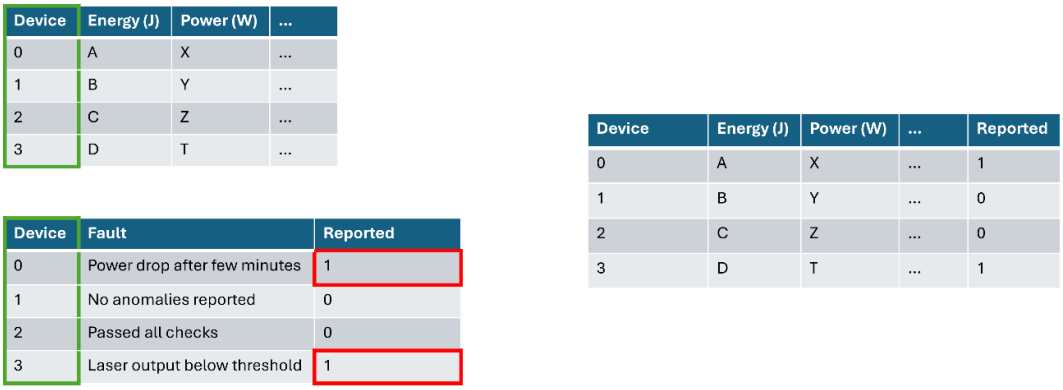


Figure 2.1 – Example of dataset connection between test data and service reports

The link between the two datasets was established using the serial number of each device. Through this mapping it was possible to assign a new (Boolean) column to the data called "reported" with value 1 if the device appears in at least one service report, and 0 otherwise. This variable is what the machine learning model will try to predict.

A quick look at the data showed that most devices had no associated intervention, which is expected, but also means that the dataset is unbalanced (we will solve this issue later in chapter 3).

The next step was to clean this raw dataset, filter out incomplete or noisy entries and make sure the features were usable for analysis. This process is described in the next paragraphs.

2.2 Features Description

The DHRs contain results from a variety of tests performed on each device before shipment. The tests include measurements of voltage, current, energy delivery, temperature stability, signal noise, and other performance indicators.

Each DHR page corresponds to a specific test protocol, for example, one section might verify optical alignment, another might check energy output in different pulse configurations (short, medium, long), and others assess pressure tolerance, laser calibration, or safety interlocks. The values recorded in these tests are numeric and standardized, although often handwritten or scanned. From these measurements, a total of 200+ features were extracted, cleaned, and standardized. Some of these represent raw sensor values (e.g., "Output Power (W)"), while others reflect internal configurations (e.g., "Modality Code", "Pulse Duration", "Energy Setpoint"). Not all features are equally relevant for prediction: some were dropped due to redundancy or missing data, while others were selected based on correlation and SHAP analysis.

2.3 Cleaning and Preprocessing Raw Data

Once the dataset was enhanced adding the target variable (reported), the next step was to improve its quality and useability, to make it suitable for the machine learning model. Predictive models are sensitive to noise and null values in input, so it was necessary some cleaning and preprocessing before moving forward. Raw data had already a clear structure, where each row represents a device and each column a technical feature, but further analysis showed that not all rows and columns were equally reliable. The first operation was to remove rows and columns with too many missing values, with a threshold of 30%: any row having more than 30% of null values was excluded from the dataset. In this way only the entries with usable information were kept, as keeping incomplete records would have introduced uncertainty and possibly distorted the analysis. In a similar way, columns with more than 70% of missing data were also removed. While true that this action reduced the dimensionality of the dataset making the next phases of analysis more efficient and focused, on the other hand it introduces a new challenge, considering that the final dataset will be used to train the AI model, and a very small dataset might not be a good fit for this purpose, even if it is clearer than the original raw data. In the chart below it is shown the comparison of the discarded feature (in red) because of null values and the ones that had been tolerated (in green).

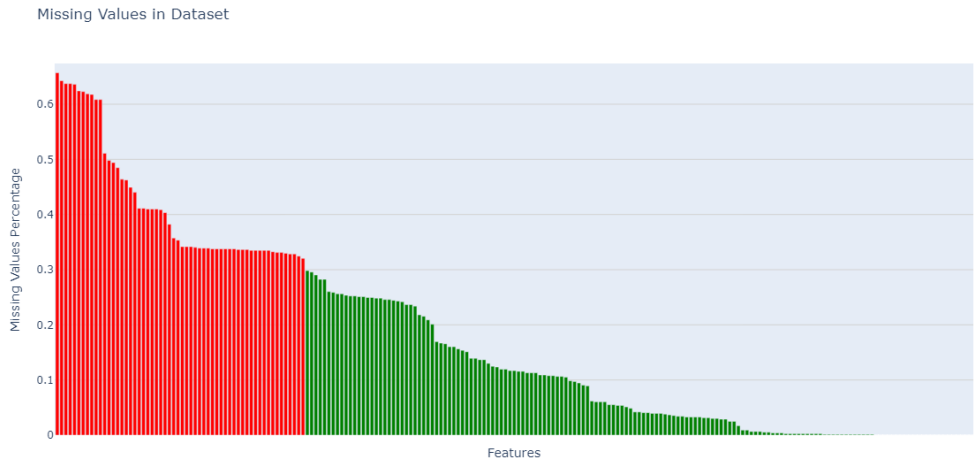


Figure 2.2 – Distribution of missing values across dataset features

Visual tools were used to better understand the distribution of missing values, and to highlight that some features had the highest number of gaps, as confirmation that most of the useful information was concentrated in a smaller set of columns. In support of this, the pie chart below shows the proportion of features falling into different categories of missing data. Interestingly, 43,9% of the features had less than 10% of missing values, suggesting that a large portion of the dataset was already well-structured. Moreover:

- 14.5% of features had between 10% and 20% missing values, which can generally be handled with simple imputation.
- 14% had between 20% and 30%, which might require more advanced techniques.
- 15% had between 30% and 40%, where caution is needed as this absence of data may affect reliability.
- Only a small fraction of features had over 50% missing data, confirming that most variables preserved a significant amount of usable information.

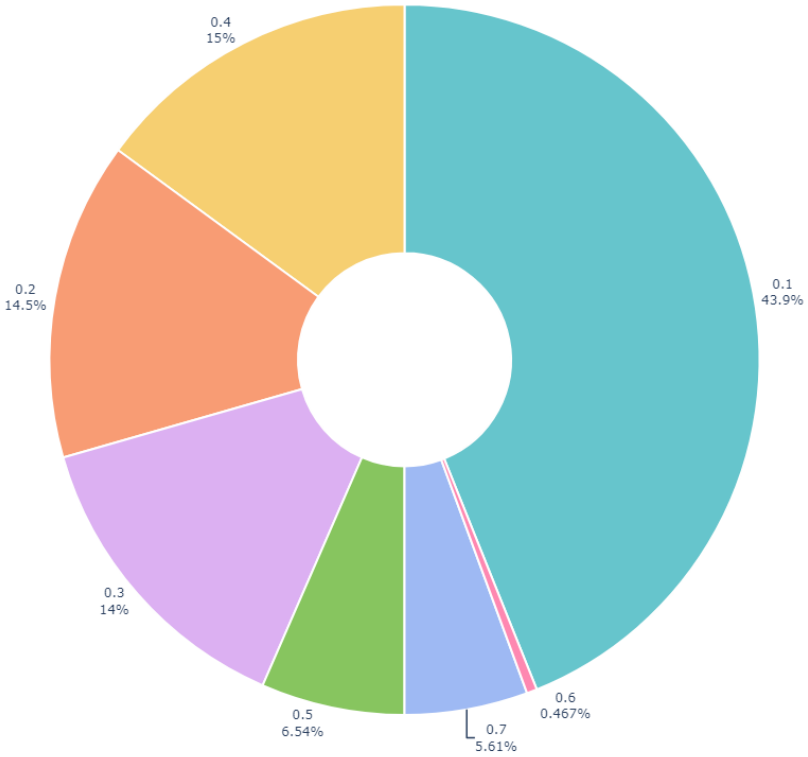


Figure 2.3 – Feature distribution by percentage of missing values.

Based on this it was possible to refine the dataset without removing too much potentially valuable data, in fact, the thresholds used for both rows and columns were not arbitrary, but they were chosen to balance the need for both data quality and quantity.

The heatmap below shows the distribution of missing values across the database, with missing entities highlighted in yellow: it gives a quick intuitive overview of how some columns, or rows, were essentially unusable due to the high percentage of null values. Columns that appear mostly yellow clearly contain little or no useful information and keeping them could even reduce the model's performance during training, by adding excessive noise.

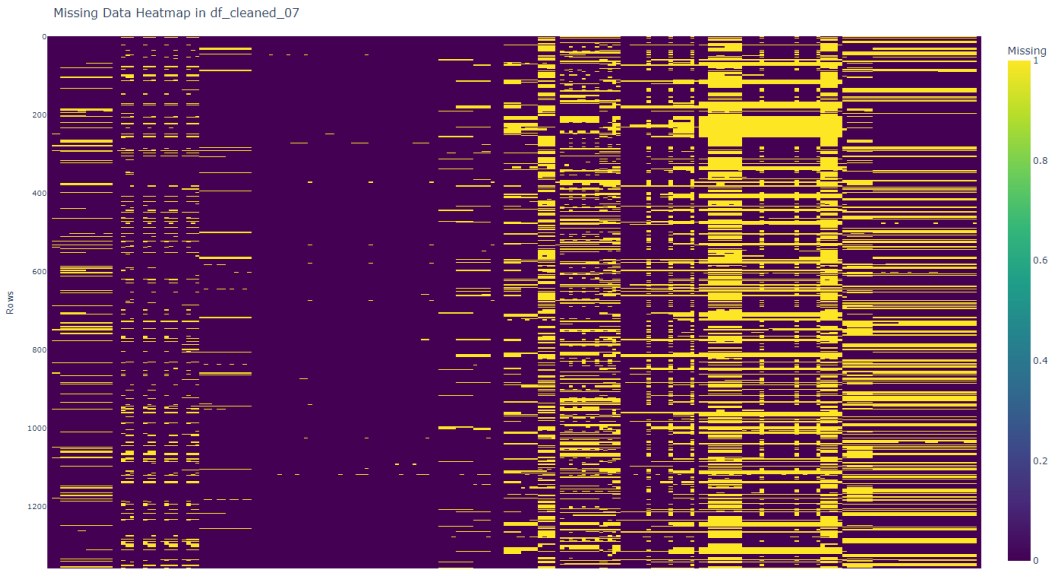


Figure 2.4 – Heatmap of missing values by rows and columns

The result of this process is a finally ready dataset which became the new reference for the next stages, clearly with fewer rows and features but now more relevant and suitable to train the predictive model.

2.4 Understanding Feature Relevance for Predictive Models

Before building a predictive model, it was necessary to identify which features carried useful information and which were not connected to the target variable (the likelihood of failure). Since the dataset contained over 200 technical parameters, it was impractical to perform a direct features selection based on domain knowledge, instead a combination of visual tools and statistical techniques were used to better understand the relationships between each feature and the binary output. This section splits the process into three parts:

- Global correlation analysis
- Statistical filter
- Distribution plots

2.4.1 Correlation matrix

To begin the feature selection process, a correlation matrix was calculated from the numerical features as the column of the dataset and the target variable *reported* that indicates whether a device experienced an intervention or not. Remember that *reported* is transformed into a binary variable even though it comes from a percentage of confidence, it simply becomes 0 or 1 depending on whether this value falls below or above the threshold. But since now it is effectively a binary variable and most features are continuous values, point-biserial correlation was evaluated as the appropriate metric to use in this case, as this method is commonly used to measure how strongly a continuous variable is associated with a binary outcome. Each value spans from -1 to 1, where values closer to the extremes indicate stronger relationships. A positive coefficient means that higher values of a feature are associated with reported failures, while a negative coefficient stands for the opposite. After computing the correlations, a bar chart was generated to visualize the intensity of these relationships.

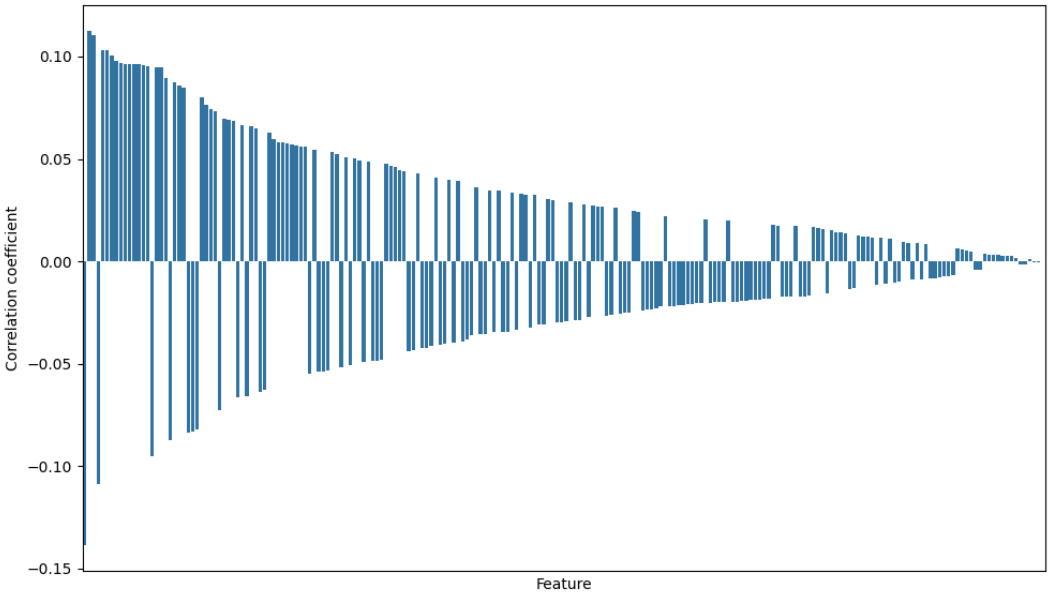


Figure 2.5 – Feature correlation with the target variable *"reported"*.

Most features showed only a weak correlation with the outcome, but a smaller set emerges with higher values above the threshold that was set to 0.05. With this visual tool it was possible to

highlight which parts of the dataset were most likely useful for predictive modeling and also raised awareness of how noisy or informative raw data really was

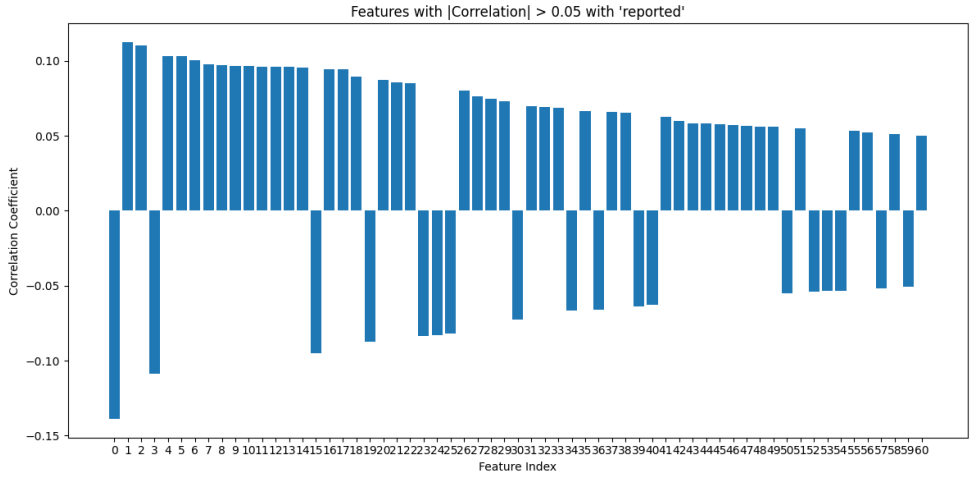


Figure 2.6a – Selected features with |correlation| > 0.05 against *"reported"*.

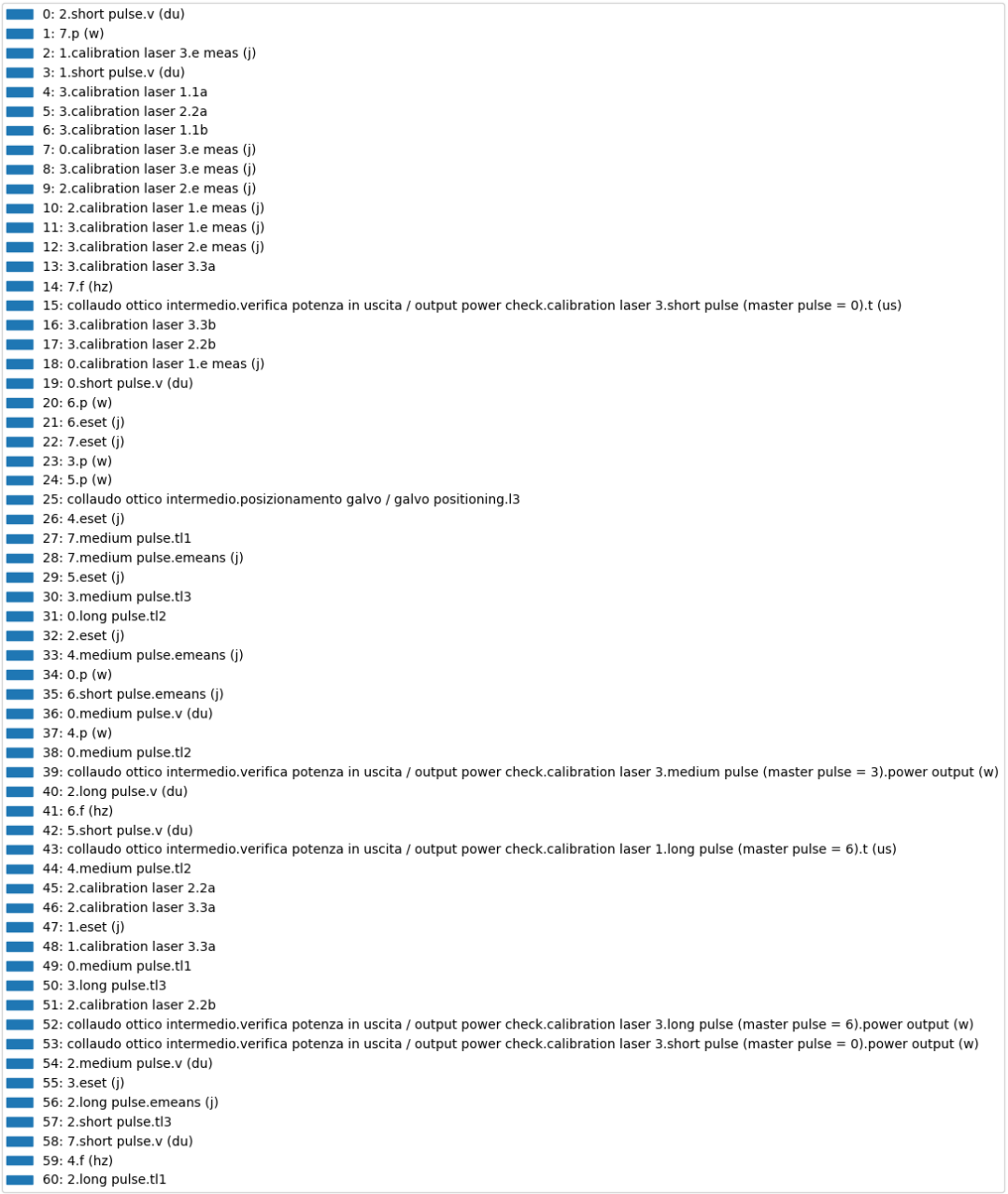


Figure 2.6b – Legend of selected features with |correlation| > 0.05

2.4.2 Feature Selection and Point-Biserial Correlation

The objective now, after having a visualization of the correlation, was to filter the dataset and keep only the most promising features for model training, and to do it following a balancing principle: reducing the number of variables to avoid noise and overfitting, without discarding useful signals too early, as some features might have a *non-linear* influence on the outcome. This particular part revealed interesting results and will be shown in chapter 4.

To do this, a point-biserial correlation was calculated, as mentioned above. This type of correlation is a reliable way to quantify how strongly a numerical value is associated with the binary outcome, the *"reported"* variable. Beside the correlation coefficient itself, the p-value was also computed for each feature to test the statistical significance of the observed relationship.

Features were kept when they met two criteria:

- Absolute correlation coefficient > 0.05, that represents a minimum strength of association.
- P-value < 0.05, to be more confident that the correlation is less likely to be due to random variation.

The result of this process was a list of features that were considered to be statistically significant, that were ready to be the base input for the predictive model.

To better understand what happened here, a different bar plot was used to visualize the top features sorted by their correlation coefficient, to illustrate which parameters (such as optical energy, pulse duration, calibration deviations etc.) were more likely associated with a service report. Some of the variables had a more obvious physical interpretation, as the relative causation seems reasonable, while others might represent hidden causation with the outcome, in a less obvious but still coherent way. While the statistical approach doesn't capture non-linear relationships or interaction effects (which will be handled by the model itself), in the next phase, it provided a sound starting point. The features selected at this stage were now ready to be used by the AI model, as they showed both statistical relevance and a reasonable connection towards the *expected* behavior of the devices.

Some features were discarded at this stage due to weak or non-significant correlations. For example, internal timestamps, serial identifiers, and batch codes showed no statistical association with the failure outcome, as expected. Others, such as ambient temperature at the time of test or internal scan IDs, displayed inconsistent patterns across samples, making them unreliable for prediction. These values may still carry latent information but considering them would likely increase noise and reduce model generalization ability.

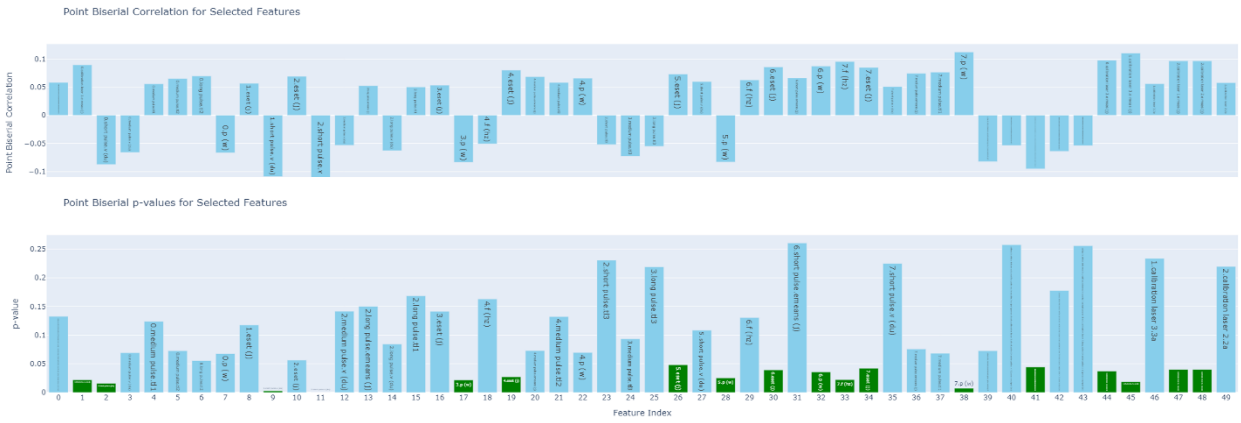


Figure 2.7 – Point-biserial correlation and p-values for selected features

2.4.3 Visual Comparison Between Classes

After selecting the most statistically relevant features, the next step was to look at them through a different point of view, using again visual tools to interpret how the features behave across the two classes: devices that reported an issue and those that did not. The goal here is not to perform a further selection of parameters and neither to discover new relationships, but only to have visual validation of what we already discovered. For this reason, several KDE plots and histograms were generated for a subset of selected features.

KDE (Kernel Density Estimation) plots give a smooth representation of a variable distribution. In this case we overlapped, in the same graphs, two KDE of the same variable in the two outcome cases: “reported” and “non-reported”. When the curves of the two classes are clearly separated, it means that the variable has a certain influence on the outcome, as it may bring useful information, but when the two curves overlap for the most part, the feature is less likely to be relevant for the prediction.

Histograms were also used in a similar way, but in this case, they were used to highlight the frequency of certain values, and the quantification of the comparison, plus they helped identify features that could affect model performance, like:

- skewed distributions
- extreme values
- noisy anomalies

In some cases, the histograms confirmed what was already known: some features showed higher values or broader distributions in the "reported" group, while others appeared to follow a totally different pattern. As mentioned before, these observations are not enough to draw solid conclusions on their own and they do not provide any scientific evidence of the direct correlation, but they serve as a visual frame of the supposed linear relations. At the end, the main use of the visual tools is to act as a "sanity check".

The table below presents the ten most statistically relevant features selected using the point-biserial correlation coefficient, a method used to assess the strength of association between a continuous variable (each feature) and a binary outcome (the "reported" label). For each feature, both the correlation value and its associated p-value are reported. While the absolute correlation coefficients are relatively modest—ranging from approximately 0.13 to 0.16—this is entirely expected in real-world industrial datasets, where failure mechanisms are often influenced by multiple interacting factors rather than by any single variable. Nonetheless, the low p-values (all below 0.001) confirm that these associations are statistically significant and unlikely to be due to chance. The features include energy measurements, pulse timing values, and calibration parameters, among the others, all of which are logically connected to the device's behavior during testing. This supports the idea that failures can be predicted, at least in part, from certain patterns or anomalies observed in these technical parameters.

Feature	Correlation	p-value
2.measured energy (j)_1.6	0,16	0.0000
5.short pulse.tl3_259	0,16	0.0000
7.medium pulse.tl3_544	0,15	0.0000
4.long pulse.tl1_673	0,15	0.0001
4.long pulse.tl2_673	0,14	0.0001
3.calibration laser 1.1b_3.58	0,14	0.0001
2.short pulse.v (du)	-0,14	0.0001
1.energy photodiode a	0,14	0.0002
3.calibration laser 3.3a_3.546	0,14	0.0002
2.long pulse.tl1_646	0,14	0.0002

Table 2.1 – Top 10 Features by Correlation with Reported Failures

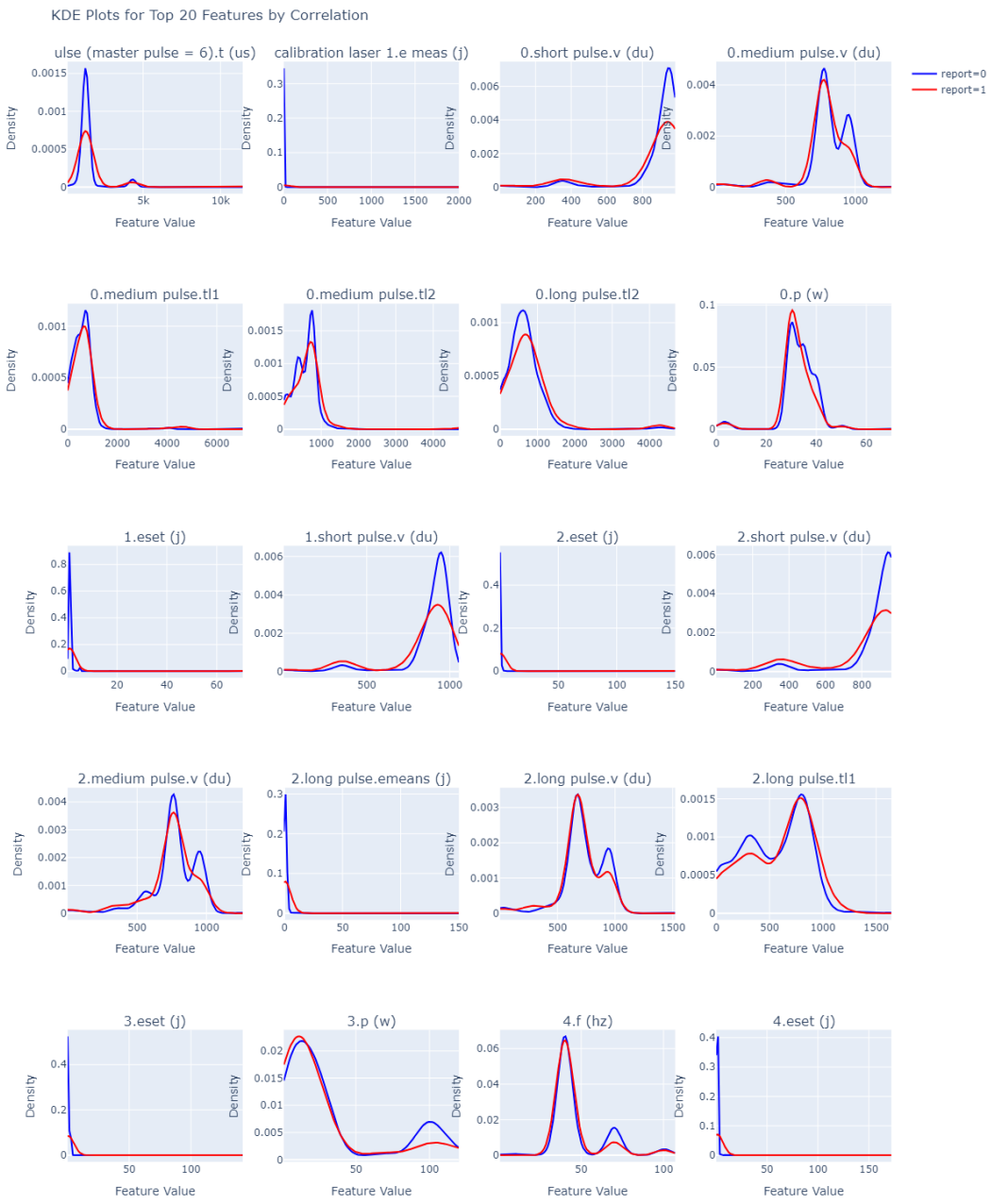


Figure 2.8 – KDE Plots for the top 20 features ranked by correlation with *"reported"*.

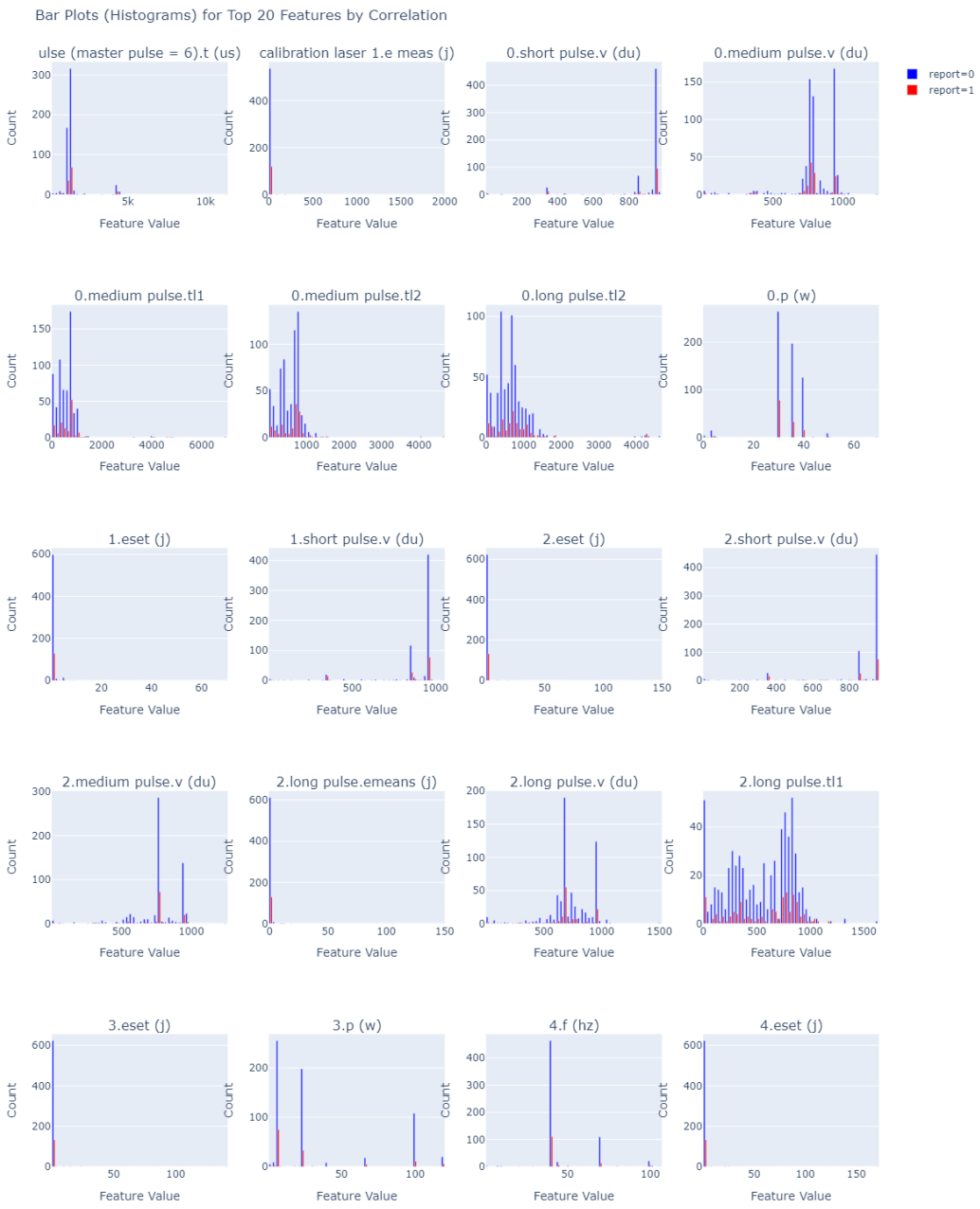


Figure 2.10 – Histograms of the top 20 features ranked by correlation with *"reported"*.

Among the features analyzed, one that really helps visualize the difference between the two classes is "pulse (Master Pulse = 6).t (us)", which basically tells us how long the long pulse lasts during a specific test. To see how this value behaves, we looked at it using two types of plots: a KDE curve and a histogram.

The KDE curve gives us a smooth idea of where the values tend to concentrate. Both “reported” and “not reported” devices peak around the same value, roughly 1000 microseconds, but with different intensities. Devices that didn’t show issues (“not reported”) are much more common in that area, while those that later had problems (“reported”) show up less often. So, even though both classes overlap in the same region, one is clearly more “crowded”.

The histogram makes this even easier to understand. It shows that around 1000 μs, more than 300 devices in the “not reported” group had this value, while only about 70 of the “reported” ones did. So that particular range is typical of devices that worked fine.

In short, even if the values seem similar at first glance, the number of times they appear makes a big difference. Qualitatively, KDE gives us the shape of the distribution; quantitatively, the histogram tells how many times that value showed up. Together, they helped us spot patterns that might not be obvious from the numbers alone, and that was very useful when deciding which features are actually meaningful for prediction.

2.5 Importance of Data Quality in Predictive Analysis

As discussed in the previous paragraph, the initial dataset did not start out as a clean one and certainly never became perfect either at the end, but it served its purpose as expected. In real world scenarios, when dealing with technical documentation, the goal is not to have a completely flawless dataset, the real challenge is to understand what kind of data can be kept, what needs to be removed and how to make the most of what remains. In fact, at the end of the cleaning process there were still some empty fields, odd values, unexpected distributions etc. But the dataset was intended to be usable, even though not picture-perfect. Another important aspect to keep in mind is that data did not come from a controlled database, but from a database populated through extraction of information through AI with OCR and language models involved, where errors may happen. It is true that the extraction pipeline overall worked fine, reaching out a 95% accuracy in recognizing values, but some errors would still be present.

There were cases where a value was shifted to the wrong column or entirely missing, though these issues did not compromise global quality of the dataset, but they highlighted that even high-accuracy systems need validation steps. For example, a value that is reported as “350”

instead of “3.50” is a small mistake from the point of view of AI, nevertheless has a major impact if it results in a huge outlier. Also, some of the variables were on a different level of interpretability: some were easy to understand and intuitively relevant, while others were harder to interpret. Anyway, they still showed consistent patterns across the two classes, for instance, some features were instantly understandable, and some just were not. Still even the less obvious ones, when they showed constant differences between machines with or without failure, could bring a useful signal for the model. At the end of the process, the resulting dataset *data_cleaned* included just what were considered the most meaningful features, all filtered and aligned to the same consistent structure: it was not a very large dataset in terms of size, but it was a solid one in terms of quality, as most of the noise were removed and features that carried valuable information were kept.

This last step of data preparation phase completes the foundation for the next stage of the project: with the cleaned data ready the focus now moves to building and training the model and testing its ability to predict future intervention based on historical signals.

Chapter 3 – Developing and Training the AI Model for Failure Prediction

3.1 Overview of the Failure Prediction Model

The previous chapter described how technical documentation was processed and cleaned to generate a reliable dataset. Now that the data was ready, the focus shifted to the actual prediction task: using historical test parameters to estimate the probability that a device will experience a malfunction. This paragraph presents an overview of the predictive model used in the project.

The main idea behind the model is to learn from past patterns. During manufacturing and quality control, each device is tested under a wide set of technical conditions: energy levels, power threshold, pulse duration, temperature readings and many other indicators. When this data is combined with historical information about whether a device later required service, it becomes possible to train a machine learning system to identify which patterns are associated with a higher risk of failure, and how important they were towards this prediction. The prediction task was set as a binary classification problem, where the goal is to assign each device a label:

- 1 = reported (the device had at least one documented service event)
- 0 = not reported (no issues had happened)

This label was introduced earlier as the target variable, and it becomes the reference against which the model learns. It's important to note that this is not a real-time prediction system, as the model is trained *offline*, using historical data. This means that it was not designed to react to live signals from machines in production and dynamically adjust its weights, but instead to act as a decision support tool. Given the tabular nature of the input data (numeric test parameters) a **Random Forest** classifier was chosen for the first iteration of the model. This algorithm was appropriate for this scenario because it can handle the following:

- Features with importance differences
- Potential non-linear relationships
- Missing values
- Without requiring extensive feature scaling

Another advantage of Random Forest is interpretability: by analyzing feature relevance coefficient, it's possible to understand which variables contributed the most to the prediction and which less, to make it easier to communicate results to the team. The model does not directly give a class label as an output. Instead, it produces a probability score that represents the estimated risk of failure for each device. This probability is then compared against a threshold: if it exceeds the threshold, the device is classified as "reported"; otherwise, it is marked as "not reported". This design allows flexibility, as the threshold can be adjusted depending on how conservative the system needs to be.

As explained in Chapter 2, the selection based on statistical relevance and correlation, returned a final dataset of **214 features**, where each row corresponds to a single device (with not more than 30% of missing values) and the data was shuffled and stratified to preserve class balance during training and testing. The model was trained using scikit-learn, and performance was evaluated with common methods such as accuracy, precision, recall and F1-score.

File ID	Galvo L1	Galvo L2	Laser1 Short Pulse Power (W)	Laser1 Medium Pulse Power (W)	Laser2 Short Pulse Power (W)	Laser2 Long Pulse Power (W)	Pulse 0 Energy (J)	Predicted Probability of Failure
CYH0085-0121	12345	23456	41.2	38.7	42.0	39.4	0.86	**0.72**
CYH0414-0220	13211	24567	40.9	39.0	41.5	40.1	0.91	**0.18**
CYH0882-0521	12789	23891	42.1	37.6	39.9	38.4	0.79	**0.53**
CYH1130-0720	12456	23678	43.5	40.2	40.6	37.9	0.92	**0.07**
CYH1480-0920	12890	23123	39.8	38.3	43.1	39.7	0.84	**0.62**
CYH2076-0722	12111	22999	40.3	36.5	42.3	38.9	0.89	**0.44**
CYH2784-1221	12678	23345	38.7	37.8	40.2	37.2	0.83	**0.27**

Table 3.1 – Example of model prediction output (reduced) with feature values and failure probabilities.

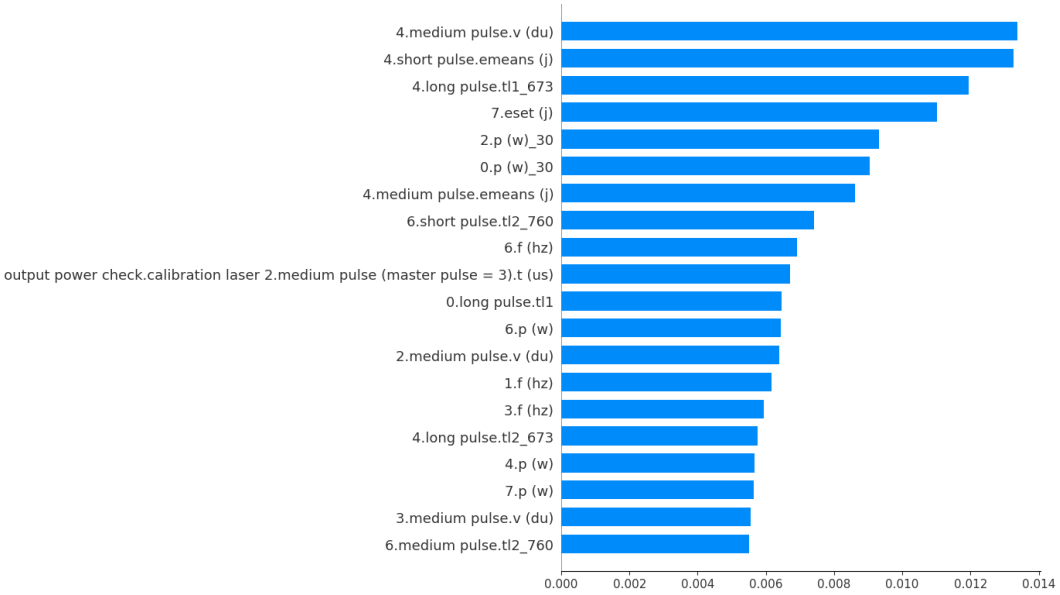


Figure 3.1 - Feature importance scores from the Random Forest model.

3.2 Selection of the Appropriate AI Model

With the dataset now cleaned and structured, and a subset of meaningful features identified through correlation analysis and visual study, the next step was to choose a predictive model tailored to the specific characteristics of this task and its assets. The goal here was to estimate the likelihood of future failures, as mentioned earlier, and this meant selecting an algorithm able to:

- Work with tabular data,
- Extract patterns from noisy input,
- Handle features of different types,
- Not require a huge amount of fine tuning.

Summarizing, the dataset built before was composed of numerical values extracted from DHRs, with over 200 features (of different scales, missing values still present in some cases, within a threshold of tolerance, and a target variable with unbalanced distribution).

These conditions immediately made some algorithms less attractive than others: for example, logistic regression and Support Vector Machines usually require normalization of the input data

and are sensitive to feature scaling. They also struggle with capturing complex and non-linear relationships unless explicitly engineered into the model. Of course, these methods perform very well in controlled scenarios or with strongly engineered features, but they lack the flexibility to adapt to more chaotic real-world data such as this one.

Several studies in the literature confirm this distinction in performance under similar conditions. For instance, a large-scale benchmark study by Couronné et al. (2018) comparing Random Forest and logistic regression across 243 real-world datasets found that Random Forest outperformed logistic regression in approximately 69% of the cases, with an average improvement of 0.029 in accuracy and 0.041 in AUC. This supports the idea that Random Forest is more robust when dealing with many weakly informative features or with datasets where the underlying relationships are not strictly linear.

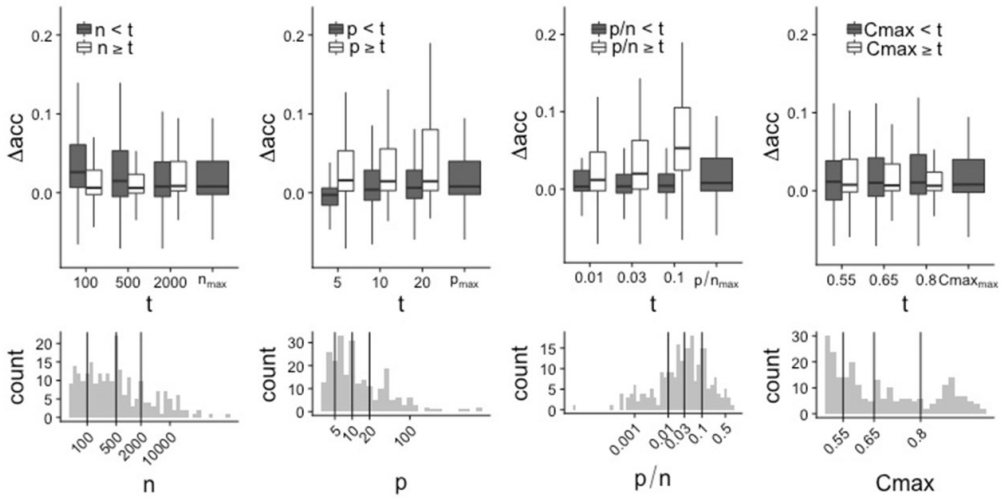


Figure 3.2 – Comparative performance of Random Forest and Logistic Regression under varying dataset conditions. Adapted from Couronné et al. (2018), *BMC Bioinformatics*.

Similarly, **deep neural networks** were considered among the alternatives. These models have shown excellent results in many domains, but they are not always the best choice, especially when the dataset is relatively small. Training a deep neural network from scratch requires a significant amount of data and computational resources, and even shallow architectures tend to generalize poorly in low-data systems. In fact, studies such as Sewak et al. (2018) show that Random Forest models can outperform shallow DNNs in terms of generalization and training

stability when data is limited, as in this case. Another known limitation of deep learning in this kind of application is the lack of interpretability. And this is not a small issue, as interpretability in the industrial context affects directly the usability of the model.

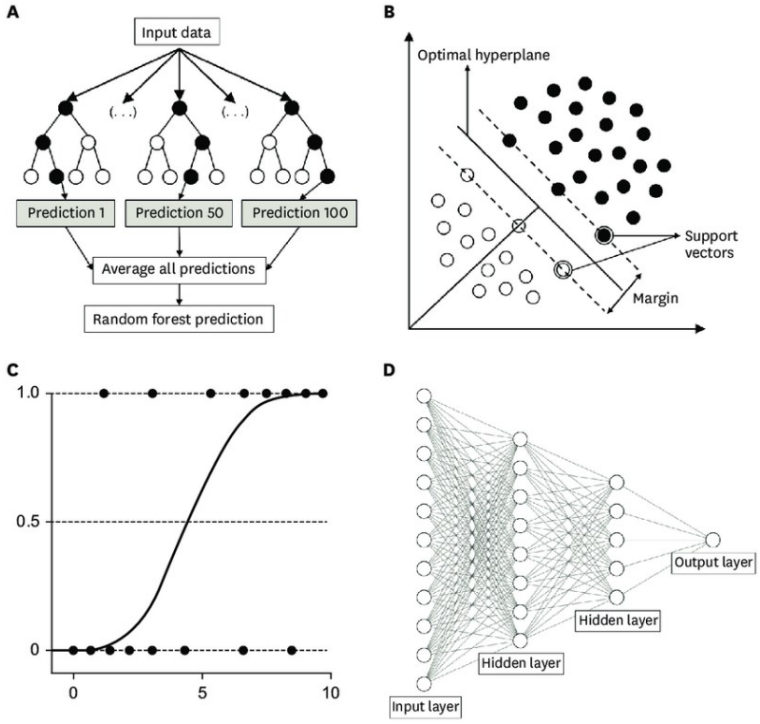


Figure 3.3 - Comparative architectures of Random Forest (A) and Deep Neural Network (D).

For example, logistic regression provides clear coefficient weights, but struggles when dealing with model nonlinearities, on the opposite, DNNs can understand complex interactions, yet are difficult to read, especially in multi-layer architectures. Random forest offers a compromise that fits this project, because it handles nonlinear patterns and still allowing internal study and inspection of how individual features contribute to a prediction.

In this project, it was essential to understand which variables influence the outcome and how, in order to extract insights and share them with engineers working on the devices. DNNs often operate as black boxes, making it difficult to explain predictions and justify model decisions. This would have made the results harder to interpret and less useful in practice, especially in a collaborative, technical environment. On the contrary, Random Forest turned out to be an

excellent match for the scenario: this algorithm creates multiple decision trees and combines their predictions to make a more robust, yet generalizable, classifier.

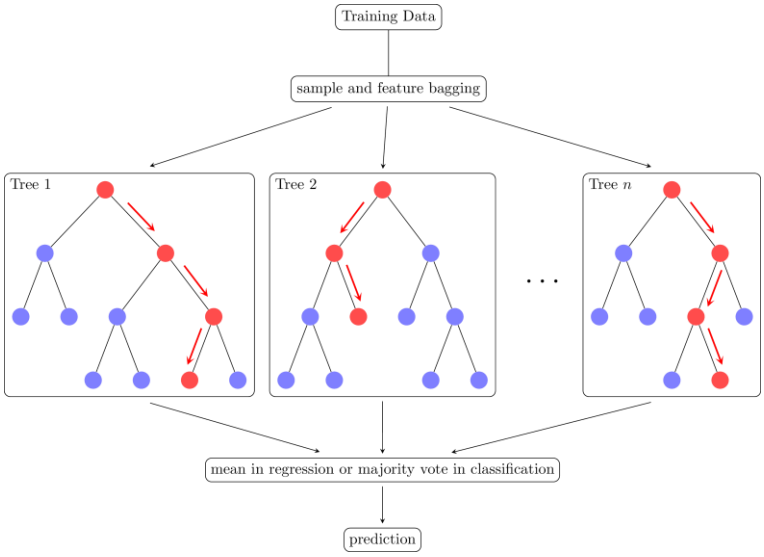


Figure 3.4 - Schematic representation of the Random Forest algorithm. Adapted from Biau & Scornet (2015).

It does not need the data to be normalized or standardized, and it is relatively insensitive to outliers and missing values, goes without saying that in real-world data extractions like this, it's a very valuable characteristic. In this context outliers are not rare, especially if we take into consideration that the values are extracted with AI, which is reliable but not immune to errors. It can also handle non-linear relationships between features, which was especially important because some of the test parameters might interact in complex (non-linear) ways that are not evident from simple correlation analysis.

Another relevant advantage of Random Forest is its interpretability. Each split in the trees can be traced back to a specific threshold applied to a feature, and the model returned a ranking of feature importance showing which variables most strongly influence the prediction. This phase was very important for the project, to evaluate the model's performance and also to extract useful insights from the data. For example, if certain laser calibration values or energy levels appeared very frequently as top predictors (meaning that in the decision trees, the same variable conditions lead to report = 1), this could guide engineers to closely monitor the retrieved critical test conditions. In addition, the model's output is a probability score rather than a fixed label,

which was appropriate considering that the threshold for classification at every node of the decision trees could be adjusted depending on operational needs. For instance, we can imagine a scenario where we need to be more conservative in identifying devices at risk because the failure cost is particularly high in some situations. This particular skill introduces a flexibility that simpler models don't always provide out of the box.

Model limitations and interpretability trade-offs

Even though the Random Forest offered a good balance of performance and transparency, they still present some limitations. As the number of trees grows, the interpretation of the whole model becomes less intuitive, despite using tools like SHAP or feature importance rankings.

Their predictions are not entirely traceable back to simple logic, but they remain accessible to engineers who need to validate decisions on technical grounds.

Model	Strengths	Weaknesses	Complexity	Interpretability
Logistic Regression	Simple, fast interpretable, robust on clean data	Poor with non-linearity, requires normalization	Low	High
Deep Neural Networks	Very expressive, scalable with data	Unstable with small datasets, not transparent	High	Low
Random Forests	Handles non-linearities, robust to noise, interpretability via trees	Acceptable transparency, slow training	Medium	Medium-high

Table 3.2 - Comparison between classification models in terms of strengths, weaknesses, complexity, and interpretability.

In these very specific conditions, every algorithm has its limits, but Random Forest seemed to be the best one. Overall, it is a combination of solid predictive capabilities, robustness and transparency, and that made it the preferred option for this application. Future iterations of the project may explore more complex models, possibly integrating ensemble methods or deep learning (if provided with a suitable dataset), but at the current stage the choice was guided by a practical balance between performance and usability in real-world context.

3.3 Training the Model with Historical Data

The next step was the training of the classification model mentioned in paragraph 3.2. To ensure a fair evaluation the dataset was split into three parts:

- training (70%)
- validation (15%)
- testing (15%)

Due to the class imbalance presented by the dataset (faulted devices were significantly fewer than the functional ones), it was necessary to introduce a stratification step to preserve the original proportion between damaged and non-damaged devices. Without this, it was possible that a random split could have easily created unbalanced sets, while stratified splitting ensures that the distribution of the target variable reported remains consistent across all three subsets, which is essential to assure a fair training path.

To handle this imbalance, the **SMOTE** (Synthetic Minority Over-sampling Technique) algorithm was applied to the training set, excluding to the validation or test sets to prevent data leakage, i.e., the model might identify validation data patterns that are too similar to those used in training. Differently from random oversampling, which simply duplicates existing instances of the minority class, SMOTE creates synthetic examples by interpolation of the ones that already existed, as the goal here was to help the model to find a perfect balance between generalization on unseen data and fitting on the known training set. In this case, overfitting would have been a reasonable risk, because the model would have been trained on a balanced dataset, where the data regarding defective devices were redundant. Anyway, this was necessary to make the model understand the failure patterns, otherwise it would have had too few samples.

This step revealed an interesting challenge: even after balancing, the model initially showed signs of conservatism "playing it safe" and underestimating the failure class. It became clear that fixing imbalance statistically is not always enough, and we understood that interpretation and threshold management are just as critical. This was a valuable lesson during experimentation.

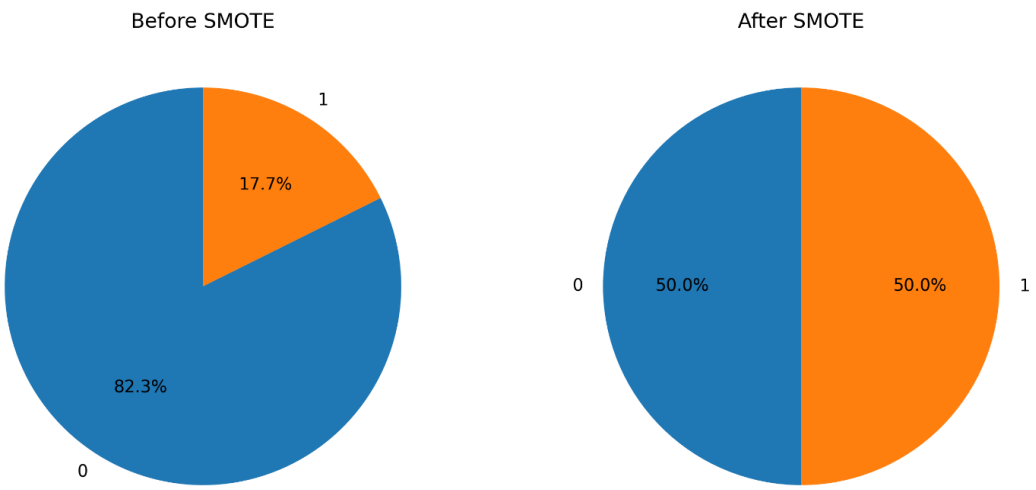


Figure 3.5 - Class distribution before and after SMOTE oversampling.

The chart shows how the original imbalance between the two classes (devices with and without reported failures) was corrected through synthetic oversampling. With a balanced training set, now the model could learn better patterns associated with the minority class (1 or "Reported").

Before training, another last modification was applied to the dataset, as we still had some inconsistencies regarding empty fields.

- For continuous variables, missing values were replaced using the median value, a method was preferred over the mean because it showed out to be more robust to outliers, which were common for several features, most likely because of misreadings or data entry errors (such as misplaced commas).
- For category variables, the most of them related to document structure or revision codes, rare values were grouped into a single placeholder category called "other" to avoid creating a large number of dummy variables during encoding.
- **Encoding** phase: most features were already numerical and only a few columns required one-hot encoding, which converts each category into a separate binary column. This transformation was done selectively to avoid creating too many columns, as the increase of the dataset's complexity and dimensionality, (as in number of columns), was actually unnecessary.

The ML model was implemented using the *RandomForestClassifier* class from the scikit-learn library. No specific scaling of the features was needed, given that Random Forest models are

not sensitive to the size of input variables: what matters in this case is the relative ordering of the values and how they interact with the splitting thresholds used in the decision trees. Each split in a tree is based on finding the best value that separates the two classes, so the relative distribution of the data is more important than its scale. In practical terms, each tree in the forest explores different feature combinations and split points progressively separating working and failed devices. This also makes the model robust to both irrelevant variables and moderate levels of noise.

This phase of model definition and training also offered the chance to experiment with several practical trade-offs. For example, while testing higher numbers of estimators, the improvement in accuracy was marginal, but training time increased significantly. Similarly, deeper trees gave slightly better recall but at the cost of interpretability. These small decisions made a big difference in finding the right balance between performance and simplicity.

The validation set was used to monitor generalization and adjust the relative hyperparameters, that in this case were:

- Number of estimators
- Minimum sample leaf
- Random state
- Class weights
- Max depth.

Grid search was not really necessary in this scenario because of the relatively small size of the dataset; however, a few combinations of values were tested manually to understand if this could affect the performance. Given the relatively small number of training samples, especially after applying filtering and balancing, a more exhaustive search would have risked overfitting to validation data or amplifying noise. In addition, the available resources (both in terms of time and hardware) did not justify a brute-force optimization process. Instead, a few targeted combinations of hyperparameters were tested manually, focusing on parameters known to influence recall, such as *max_depth, class_weight*, and *min_samples_leaf*. This results way less systematic, yet it proved effective for our scenario: it allowed for a reasonable level of tuning without overcomplicating the process or relying on metrics alone. Manual testing also gave space for iterative reasoning, where each change was evaluated not just on scores, but also on the model's behavior in borderline cases.

In hindsight, this lightweight tuning approach was actually quite efficient for our setup, avoiding overfitting while still improving precision.

After convergence, the model was tested on the held-out test set to obtain an unbiased performance evaluation. It is worth noticing that the training process had to consider some of the domain-specific constraints discussed in earlier sections, in particular, the model was initially biased to minimize false negatives. In its first iterations it leaned heavily toward conservative predictions, often failing to detect true failures. This resulted in a low number of false positives, but at the cost of many false negatives, which was considered as an unacceptable trade-off in predictive maintenance. This behavior highlighted the need for further adjustments in thresholding and evaluation.

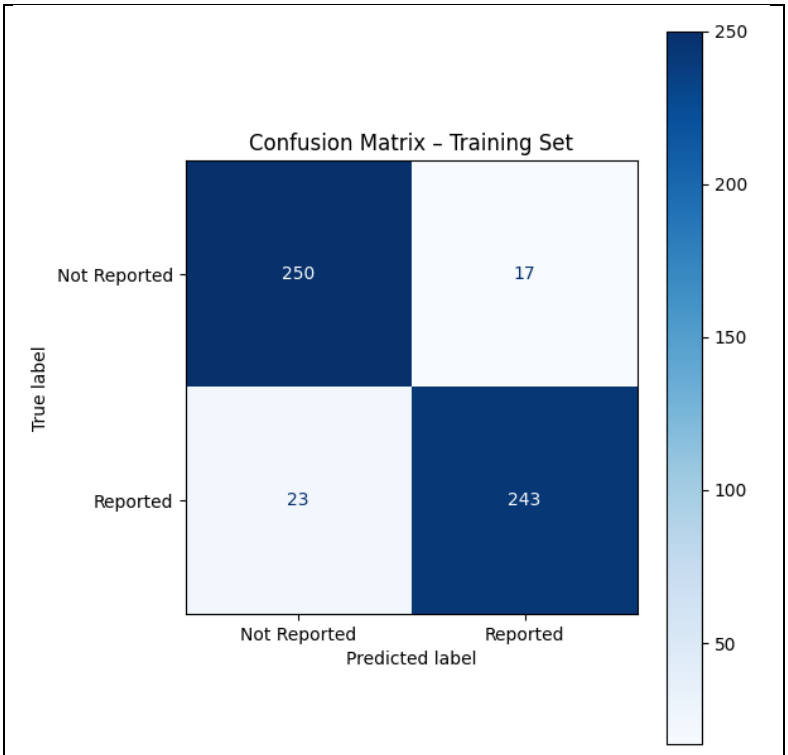
3.4 Fine-Tuning and Evaluating Model Performance

After the training phase, a preliminary assessment was necessary before performing real tests. What was evaluated here was the model's ability to generalize to unseen data and to retrieve the final information about the most influencing features. This evaluation was carried out on the test set which had been kept completely separate from the training and validation ones, to ensure unbiased performance estimation. We didn't want a score, instead the team wanted to examine how the model behaved across different situations such as dealing with defective devices, which represent the minority class in the dataset.

In the first evaluation, the model assigned relatively low probability scores to most devices, and because of the initial threshold configuration set too high, to favor precision over recall, very few predictions crossed the "reported" classification boundary. As a result, the system failed to trigger failure predictions, even when risk signals were present. The model somehow quickly learned to completely avoid false positives at the cost of ignoring most of the faults: this behavior proved unhelpful for predictive purposes, though it was aligned with the training conditions. These occurrences made it very clear that tuning a model is not just about maximizing metrics, it's about matching those metrics to the context. Here, missing a failure was far worse than over-predicting one, and this shift in perspective changed how we interpreted all the numbers.

The first step regarding the evaluation phase involved the use of confusion matrix, which is a visualization of the results of the predictions, divided into four categories:

- true positives (correctly identified failures)
- true negatives (correctly identified non-failures)
- false positives (functional devices misclassified as failed)
- false negatives (failures misclassified as functional devices)



TRAINING SET

	Precision	Recall	F1-score	Support
Not reported	0,916	0,936	0,926	267
Reported	0,935	0,914	0,924	266
Accuracy			0,925	533
Macro avg	0,925	0,925	0,925	
Weighted avg	0,925	0,925	0,925	

Figure 3.6 – Confusion matrix and classification metrics for model performance (training set).

Among these categories, given the context, the false negatives were considered to be the most dangerous: in an industrial context, failing to detect an actual fault may result in undiagnosed malfunctions, that in turn could translate into unexpected downtime and quality issues.

To provide a more complete overview, the generation of a classification report was implemented, with metrics like precision, recall and F1-score for each class. Recall in particular

was important because it can detect actual faults while metrics like accuracy may seem good due to the fact that half of the (training) dataset was labeled as “functioning device”. Trying to face this trade-off, we started experimenting with alternative classification thresholds, deviating from the default 0.5 cut-off probability. The objective was to recover at least a portion of the missed failures and still keep the number of false alarms within an acceptable range.

This phase required a very delicate balance, too aggressive a threshold would degrade trust in the system, too cautious a threshold would leave the prediction actually useless. This is why, at the cost of slightly more false positives, efforts were made to optimize the model toward a higher recall on the positive class (failures). In parallel with threshold adjustment, through validation set, it’s been conducted a systematic hyperparameter optimization. The model has been trained and evaluated through a combination of key parameters (seen in previous section). The best performing configuration was:

- *n_estimator = 100*
- *max depth = 20 min_sample_leaf = 5*

Special attention was paid to class weight parameter because this represents the level of caution towards false negatives, so these special hyperparameter values have been experimented manually. A “balanced” set was not ideal because of the dataset disproportion in favor of functional devices, and SMOTE was not used in the validation set. At the end, the best performances in terms of F1-score was given by this setting:

- *Class_weights = {0: 1, 1: 5}*

This represents the toll attributed to an error on the specific class, and respecting the reality of the application, a heavier weight was given to the errors we absolutely wanted to be avoided.

In any case, a model that could not predict malfunctions would have been totally unuseful, especially considering all the effort made until this stage. As for the training set, it’s important to clarify that these metrics may not seem informative enough on their own. In this case, values almost appear to be perfect, which might seem suspicious at first glance. However, this is not unusual in machine learning, especially when working with ensemble models like Random Forest. We need to consider that we are now evaluating performance on the same data the model was trained on, so these results reflect the model’s ability to fit to known data, still does not talk

about the generalization skill of this model, and that's why these numbers are not meant to serve as evidence of good real-world performance.

Instead, the aim of this table is just to illustrate how each metric works and what they measure. The true generalization ability of the model will be evaluated later in Chapter 4, with the validation and test sets that contain data that the model has never seen, so that they are better fitted to simulate an actual practical scenario.

The next issue that has been faced was the explainability of the decision. In context like predictive maintenance, understanding that the model works is not enough, for this reason **SHAP** (Shapley Additive exPlanation) was applied to interpret the model's choices. SHAP is a framework based on game theory that explains the output of any machine learning model by attributing to each feature a contribution score for a given prediction; at the end, these scores will tell how much each feature pushed the prediction towards class 0 or 1.

It follows a generation of a SHAP summary plot that visualizes the average absolute SHAP values across test samples. The plot ranks the most influential features and also shows whether high or low values for that feature tend to increase the predicted risk of failure.

For example, let's say that one of the most relevant features is the energy measured during the short pulse calibration: the SHAP plot could reveal that low values of this element lead to the failure class (1) end high values lead to non-failure class (0).

Here is an example from the dataset showing some of the feature values alongside the SHAP values assigned to them:

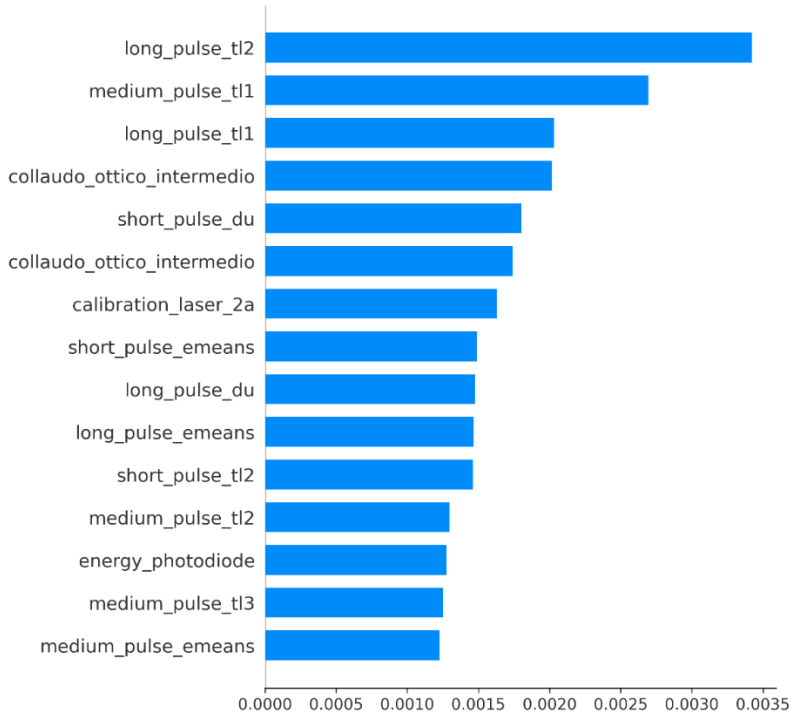


Figure 3.6 – Mean absolute SHAP values for the most influential features.

This kind of visualization makes the model's internal logic accessible even to non-technical users. Some features that had already shown up as important in the correlation and point-biserial analysis were confirmed by SHAP, like Pulse energy, Output power, Galvo positioning etc. Other features emerged more unexpectedly, as confirmation that Random Forest can identify interactions that are not visible through univariate statistics alone. This assessment will be covered in chapter 4.

This cross-check between different analysis levels really validates both the interpretability and the consistency of the pipeline. The validation set was continuously used during training to fine-tune the model and monitor its generalization. The fact that performances on the test set stayed aligned with what was observed during validation indicated that no major issues due to overfitting have happened. An evidence of overfitting could have been a dramatic drop in performance, meaning that the model would have memorized the training data instead of learning useful patterns, and that in would no longer be able to generalize to new information. This was actually one of the reasons why we previously decided to apply a SMOTE balancing technique.

This evaluation phase served a dual purpose: it confirmed the model's quality on prediction and built confidence towards its decisions through visually clear explanations. The latter is a key point when working in real-world scenarios, where black boxes are no longer accepted and transparent and justifiable behavior is desired.

3.5 Integrating the Model into a Maintenance System

Beyond the modeling stage, a relevant aspect of any AI system is its potential after deployment.

In this context, the Random Forest model was not just evaluated as a static tool but considered as part of a wider intelligent maintenance pipeline, in order to move from a successful POC to a system that could really support operators and analysts in day-to-day decisions.

The trained model was exported using **Python**'s built-in serialization mechanism, producing a file with .pkl (pickle) that contains the entire trained Random Forest classifier, including the learned structure of the decision trees and all its parameters. By saving the model in this format, it is possible now to reload it when needed, with no retrain required. This choice is valuable in production contexts, where the model might need to be used repeatedly, most likely on new data, by systems totally different from the original development. For example, a separate application could need the .pkl model to elaborate some new different types of DHR or service reports, with no additional training; this choice is the key to having a modular AI system that is completely scalable.

Business Action: Deployment strategies

We had two options:

- the model could be exposed through **REST API**, allowing other tools to send structured input and receive back an instant prediction (about probability of failure in this case); this solution would support real-time integration, for example as soon as DHR is processed, the system could immediately make a risk prediction and flag the document for inspection.

- Otherwise, a **batch-based** solution was taken into consideration, in this case the model is executed once in a while on new DHRs that in the meanwhile have been accumulated in a centralized database.

From an operational point of view the expected behavior was something like this: as soon as a new document is ingested and transformed into a structured record, it would be instantly passed to a model that would return a probability of fault. This prediction can then be inspected by a human or trigger a workflow.

This would close the loop between data acquisition, model inference and business action:

Model Updates

This was another concern to take into consideration: while the initial model was trained on a fixed dataset, in a real-world setting the system should have the capability of retraining. As new DHRs are processed and more failure outcomes become available, periodic retraining is desired to keep the system updated, to adapt the model to eventual changes, on hardware revisions, or on testing protocols. This is in order to avoid losing predictive power over time and changes, and model drifting.

Usability

Even best performing models cannot be valuable unless their output is interpretable and can generate insights that can easily be accessible by relevant stakeholders, and this is where the feature importance (like the ones provided by SHAP) can also be integrated into the user interface to show that a prediction has been made and why it has been made. With the model now fully trained, tested and integrated into a deployable solution, equipped with interpretability tools, the foundation for a predictive maintenance system is operationally complete; all components from data acquisition to deployment have been structured in a pipeline that is both robust and scalable. But building a model is only one part of the equation, because what matters most at this stage is how well it performs in practice, and how its predictions can support actually real decisions.

The next chapter will then be focused on the system evaluation about its effectiveness through real-world metrics and explorations of limits and opportunities .

Chapter 4 – System Evaluation

Up to this point, the training metrics have provided a preliminary indication of how the model behaves, but only within the limits of what it has already seen. This is not sufficient to evaluate its reliability in real-world applications. This chapter presents a structured assessment of the system's performance, using quantitative evaluation and interpretability tools to extract actionable insights from realistic scenarios. The main objective is to understand how the model performs on unseen data, how it compares to alternative approaches, and what practical value it offers when integrated into the client's actual maintenance workflow. Special attention is given to two critical aspects: minimizing false negatives, which are particularly harmful in fault detection contexts, and ensuring model transparency, which is essential for trust and usability in industrial environments.

4.1 Testing the AI Model: Evaluation Metrics and Criteria

One of the most revealing stages in any machine learning project is when the model is confronted with data it has never seen before. In this case, that moment came during the validation and test phases. Both sets were held out during training, and they were not altered with oversampling techniques like SMOTE. Their purpose was to simulate real-world distributions in which failure events are rare but need to be anticipated, and measure how well the model could still recognize meaningful risk signals under these conditions.

In the first stage, the model struggled a lot when it needed to predict failures. Even with class balancing and some early tuning strategies applied, it often chose to lean on the majority class, essentially playing it safe. This behavior can seem accurate from a superficial point of view because it performs well in recognizing devices that work just fine, but in reality, failed to address the very reason the system was being built, namely, to predict malfunctions based on early signs information. This behavior can be explained through understanding the nature of the data itself: the positive class (reported = 1) had a small cardilaity and was also internally inconsistent, as not all failures looked the same, and their underlying features were often subtle or noisy. Of course this seems like an excuse , but it really made it difficult for the model to

learn a confident rule for what "failure" looks like. The result was a classifier that would rather lay on a secure side instead of taking the risk of a false alarm. Though, under these circumstances, this choice is costly. A false negative means a missed failure, i.e., unplanned downtime, customer complaints, and in general, the system does not perform the action that is asked, nullifying all the work done. On the other hand, accepting some false positives, implies the need to filter out badly-classified data, through human review. This difference in consideration led to a shift in strategy: the model had to be encouraged to be less conservative, at the cost of tolerating some more false positives. To achieve this, there was a need to change some parameters. As can be seen later, the classification threshold needed to be lowered to meet the targeted recall value. Additional sampling and weighting strategies were tested to amplify the signal of the minority class during training. Moreover, evaluation criteria were focused more on class-wise performance than on global accuracy.

Gradually, the model began to recognize some failure patterns, showing early signs of generalization beyond what it had directly seen during training. The change wasn't drastic, but it was consistent. When faults occurred, the model started to notice them, still not always, but often enough to justify its function.

Most importantly, this behavior held up not only in validation but also in testing, which confirmed that the shift wasn't just noise or chance. It was the result of deliberate design choices, applied in response to a clear operational need.

4.2 Validation and Test Results

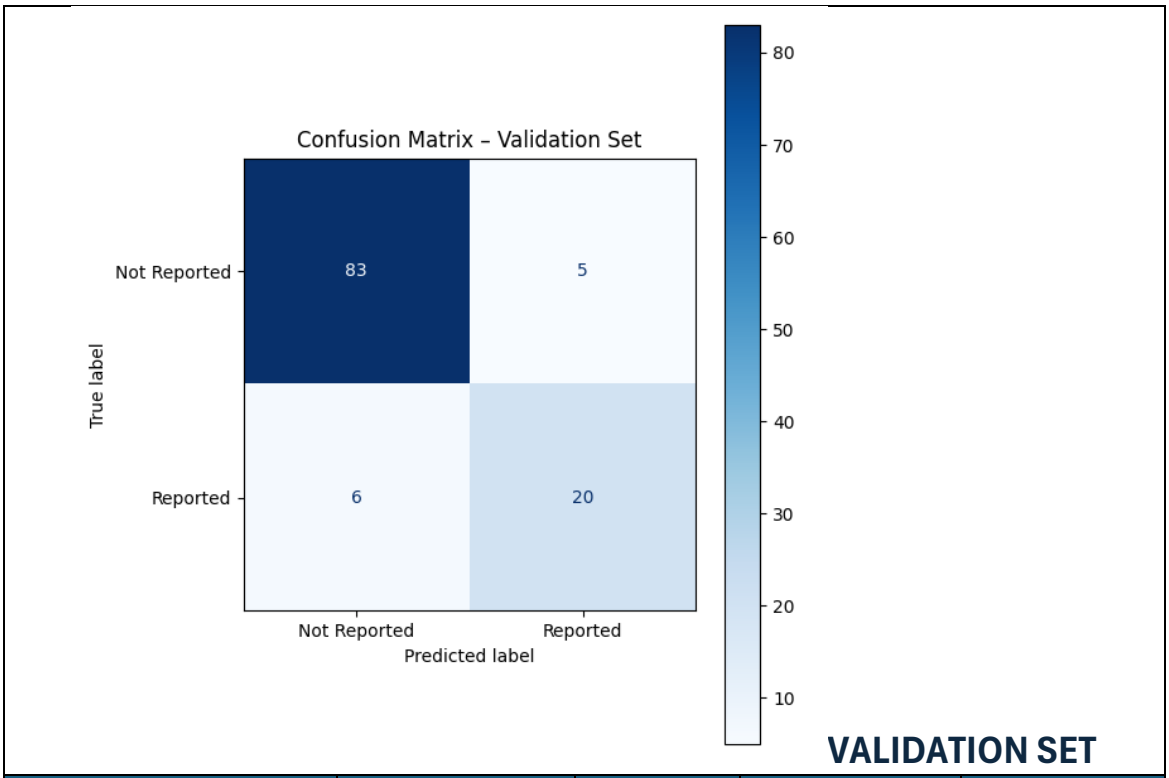
After the tuning and calibration described in the previous sections, the model was finally evaluated on the validation and test sets. These two stages represented the most important moment for assessing generalization. They offered insight into how the model behaves when exposed to new data, data that actually reflects the real distribution observed in production, where failures are rare and class proportions are unbalanced.

While the training set had been deliberately rebalanced with SMOTE to help the model focus on minority class patterns, the validation and test sets were left untouched. This was a deliberate decision to avoid boosting the model's confidence or misleading its generalization ability. In

this way, the results obtained on these two subsets are a much more realistic simulation for how the model might behave once integrated into a real maintenance pipeline.

4.2.1 Class-wise Performance Analysis

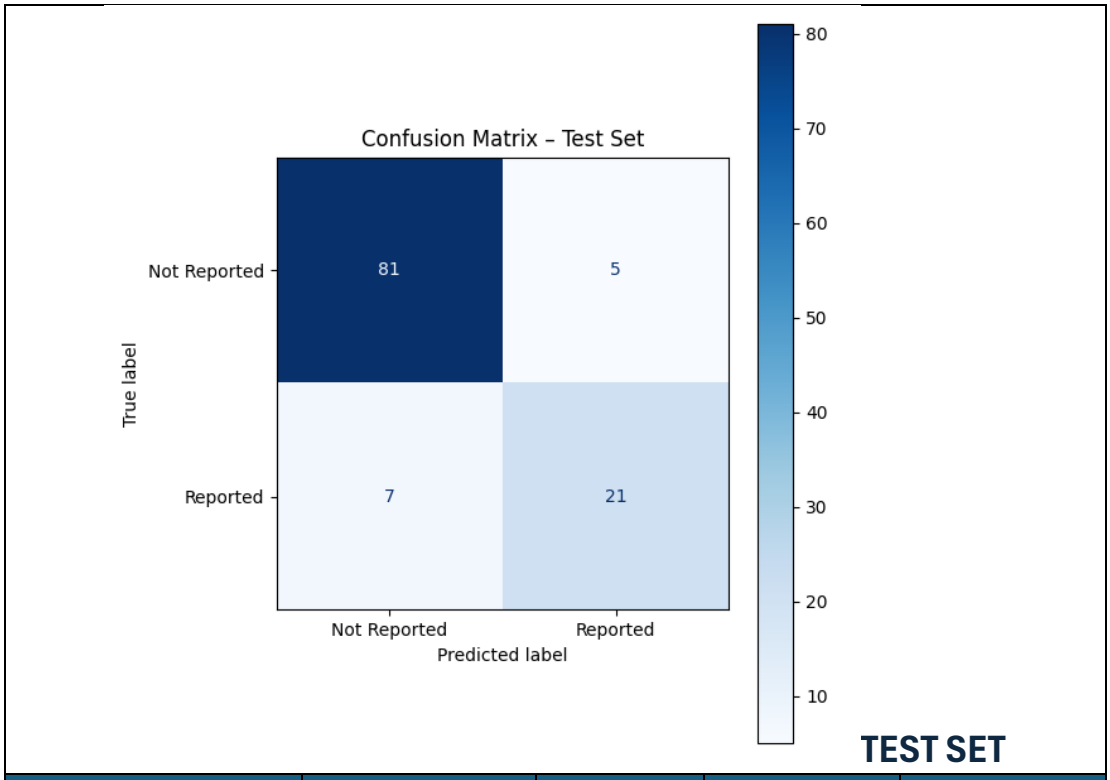
The model's predictions on validation and test were summarized using the standard classification metrics: precision, recall, F1-score. These metrics were computed for each class separately and then averaged using both macro and weighted approaches. The detailed results are reported in the following tables. In both subsets, the results show consistent behavior: the model managed to preserve a good level of sensitivity on the minority class without dramatically sacrificing precision. Recall values around 0.75–0.77 confirm that the model was effectively able to catch three out of four failures—an acceptable rate considering the complexity and noise of the data.



	Precision	Recall	F1-score	Support
Not reported	0,933	0,943	0,938	88
Reported	0,800	0,769	0,784	26
Accuracy			0,904	114

Macro avg	0,866	0,856	0,861	
Weighted avg	0,902	0,904	0,903	

Figure 4.1 - Confusion matrix and classification metrics for model performance (validation set).



	Precision	Recall	F1-score	Support
Not reported	0,920	0,942	0,931	86
Reported	0,808	0,750	0,778	28
Accuracy			0,895	114
Macro avg	0,864	0,846	0,854	
Weighted avg	0,893	0,895	0,893	

Figure 4.2 - Confusion matrix and classification metrics for model performance (training set).

The precision remained just above 0.80 on the positive class, which means that among the devices flagged as faulty, only a small portion turned out to be false alarms. From an operational standpoint, this trade-off is reasonable and acceptable. It supports a practical use case where flagged devices undergo additional checks, while minimizing the risk of missing actual malfunctions.

4.2.2 Comparison with Training Phase

To better understand these results, it helps to compare them with the metrics obtained during training. Back then, the model performed almost too well, for understandable reasons. That's not unusual: during training, the dataset was balanced artificially, so the model had an easier time learning the patterns, especially for the minority class. But things change when moving to validation and test sets, which were left imbalanced and closer to real-world conditions. Here, the model couldn't just memorize, as it had to actually understand the behavior of faulty devices in a more realistic and noisier context. Even though the metrics dropped a bit especially for the failure class, there was no sign of overfitting. The results stayed consistent between validation and test sets, and none of the scores collapsed dramatically. In fact, seeing recall values around 0.75 and F1 around 0.78 suggests that the model managed to generalize well enough, without unpredictable swings.

And that was one of the main objectives from the start: to create a fault detection system that stays stable and reliable once deployed.

4.2.3 Metric's Interpretation

It is also important to understand what these metrics actually mean in the context of predictive maintenance. In classic classification tasks, a high **accuracy** might be desirable, yet in this situation it may be misleading. A model that always predicts "no fault" would still achieve accuracy above 75%, just by exploiting class imbalance. What matters is the ability to recognize failures when they occur (**recall**), and to do so with enough confidence to avoid false positives (**precision**). The **macro average** treats each class equally and gives a balanced view and avoids being dominated by the majority class. The **weighted average**, on the other hand, reflects the real-world distribution. The small gap between macro and weighted metrics confirms that the model does not overly favor the dominant class. Another subtle but important signal lies in the **F1-score** because it matches precision and recall, it highlights if the model is biased toward one or the other. In this case, with both metrics fluctuating around 0.75–0.80, the F1-score confirms that the classifier remains balanced in its behavior, and this equilibrium is sustained across datasets.

4.2.4 Implications for Deployment and Future Work

From a deployment perspective, the results show that the system can already serve as a preliminary screening layer in the maintenance workflow. It's not perfect but it provides a probabilistic signal strong enough to prioritize further checks, especially when operating on large volumes of equipment. The recall values suggest that a significant fraction of failures can be caught early, reducing downtime and improving responsiveness.

Some limitations remain though: the number of false negatives is not negligible, and for sure future work should explore ways to reduce them without compromising precision. This may include enriching the dataset with more fault examples, improving feature extraction logic, or exploring more expressive models such as gradient boosting or hybrid architectures that combine ML with rule-based systems. In any case, the current model stands as a solid foundation for now: its behavior is interpretable, its performance is consistent, and it meets the industrial requirement of "better than chance, but not blindly confident."

4.3 Feature Importance and Explainability

One of the key strengths of the Random Forest model is its inherent interpretability. Unlike deep learning models, which often behave like black boxes, tree-based ensembles allow us to investigate how decisions are made and what features influenced them the most. In a context like industrial setting where predictions impact technical validation and risk assessment, understanding why a certain output is produced becomes just as important as the output itself. To enhance this interpretability, the project leveraged SHAP (SHapley Additive exPlanations), a method that assigns each feature a contribution value for each prediction. Based on game theory, SHAP offers global feature importance and also local explanations, letting us know for every single device which feature pushed the prediction toward failure and which pulled it toward normal operation, a skill that is important for debugging and for model validation.

During model development, a classic statistical analysis using point-biserial correlation had already provided an initial list of features weakly or strongly associated with the target label. But that method, by its nature, evaluates each variable in isolation and assumes linearity. SHAP, on the other hand, goes deeper: it captures interactions between variables, thresholds, and non-linear effects that are invisible to simple metrics.

This allowed us to intuitively sense the correlation and the differences of the two perspectives on feature importance, one based on raw statistics, the other based on the internal behavior or the training model. For instance, output power and pulse energy stood out in both analyses, confirming their relevance. But SHAP also revealed the high influence of time-related and pulse width features like medium_pulse_tl1 and long_pulse_tl2, which had a weaker correlation on their own but proved impactful when combined with others: the model was capturing real patterns buried in the data in addiction to simple associations. To visualize this comparison, a table was created listing the top features according to both approaches. The partial overlap reinforces the idea that the two methods offer complementary views:

Rank	Feature (Correlation-Based)	Feature (SHAP-Based)
1	measured_energy (j)_1.6	long_pulse_tl2
2	short_pulse.tl3_259	medium_pulse_tl1
3	medium_pulse.tl3_544	long_pulse_tl1
4	long_pulse.tl1_673	collaudo_ottico_intermedio
5	calibration_laser_1.1b_3.58	short_pulse_du
6	short_pulse.v (du)	calibration_laser_2a
7	energy_photodiode_a	short_pulse_emeans
8	calibration_laser_3.3a_3.546	long_pulse_du
9	long_pulse.tl1_646	energy_photodiode
10	calibration_laser_2a	medium_pulse_tl3

Table 4.1 – Comparison of top-ranked features from correlation analysis and SHAP values.

While only a subset of features appears in both lists, this divergence is not a weakness, it's a sign that the model goes beyond simple metrics. A variable might be uncorrelated when isolated, in reality it plays a key role in interaction terms. This is the case with features like du (pulse duration) and calibration steps, which gained importance through SHAP analysis even though they weren't top ranked statistically. Another benefit of SHAP lies in its ability to produce local explanations. Using force plots or dependence plots, it was possible to see how a small variation in a specific parameter, such as lower medium pulse power value, could increase the predicted probability of failure. These plots turn raw data into something actionable: a technician can visually understand why a device is flagged as risky, and which value might be outside the expected range. In one case, low power in short pulse mode consistently pushed the

prediction toward the fault class. This matched a known issue discussed with field engineers, confirming that the model was not just accurate, but also aligned with domain knowledge. SHAP made this connection explicit and measurable, helping to justify its inclusion in a monitoring system.

Finally, SHAP also helped uncover less obvious patterns. Some features showed relevance only in specific configurations, for example, energy metrics mattered only when galvo misalignments were present. Simple statistical methods wouldn't catch these patterns, but a tree-based model can recognize them without needing any extra instructions.

In conclusion, integrating SHAP into the workflow brought three major benefits:

- It confirmed and extended the insights from statistical feature selection.
- It made predictions understandable to both data scientists and domain experts.
- And it gave the system a level of transparency and auditability that's often missing in AI applications.

This step effectively closed the loop from model training to operational deployment, ensuring the system could be trusted, improved, and used as a foundation for future iterations.

4.4 Limitations and Edge Cases

Even though the model performed fairly well on validation and test sets, it still showed some clear limitations. Some of these issues come directly from the nature of the dataset, others are related to how the model works. Understanding when and why the system fails is just as important as celebrating when it gets things right, especially in critical environments like this one.

This section will summarize all the limitations and flaws experienced until this stage of the project:

One of the main challenges we faced was the imbalance between the two classes. In real data, actual failures are rare compared to functional devices. This reflects how things usually go in production, but it makes training a model much harder. If no action is taken, most models will naturally learn to predict "no fault" almost all the time. That's exactly what happened in the

early stages of this project. The model learned to play it safe, avoiding risky predictions and essentially ignoring the minority class.

At first, we used the standard setting:

```
class_weight='balanced'
```

hoping it would help the model deal with the imbalance. But the results were disappointing. So, we manually adjusted the class weights, setting them to

```
class_weight={0: 1, 1: 5}
```

to push the model to focus more on the faulty class. We also used SMOTE to oversample the failure cases in the training set. These solutions helped a bit, especially during training, but they weren't enough to fully solve the problem. When we tested the model on real, unbalanced data (validation and test sets), the skew remained, and it was clear that the model still didn't see enough variation among the failure cases to learn them properly. One direct consequence was the number of false negatives. Even in the final test phase, 7 out of 28 faulty devices were missed. That's one out of four. In practice, this means that the model sometimes said "played safe" when it wasn't the case. And in an industrial context, that can be a real problem. False positives (wrongly flagging working devices) can be annoying, but they can be double-checked. False negatives are worse: there are missed chances to prevent issues, and they can lead to production stops or even safety risks. This cautious behavior wasn't intentional. It came from the model's early tendency to always trust the majority class, because that's what it saw the most during training. Even with class weights adjusted, the model only started to react to risk once we lowered the decision threshold and tested other rebalancing techniques. Only then did it start to take more calculated risks and flag some uncertain but potentially important cases.

Still, the model isn't perfect. It tends to struggle in certain situations like edge cases. These are samples that lay somewhere between the two classes, where the signal isn't strong in either direction. For example, devices with calibration values just slightly out of range, or borderline energy levels. These cases don't clearly resemble known failures, but they might still be early signs of problems. The model often gets confused here, with prediction probabilities floating around the decision threshold. This highlights a basic limit of binary classification: real-world faults are gradual and messy and not clean binary labels. Another limitation is the lack of contextual information. The features we used were extracted from technical reports (DHRs),

and each sample was treated as an isolated snapshot. But in real life, failure usually develops over time. Knowing things like how long the device has been in use, how often it was serviced, or whether other similar units failed before could make a big difference. This information wasn't available on this project, but it would definitely help with future iterations. Adding more context would definitely make the model smarter and more reliable.

Finally, while we picked Random Forest partly because of its interpretability, that too has its limits. Yes, it's more transparent than a deep neural network, and with SHAP we can explain predictions to some extent. But we're still talking about an ensemble of hundreds of trees. For some users, even if a visual explanation isn't enough, they may need rules that are clear, auditable, and easy to follow. And that's not always possible, even with the best explainability tools.

The model works, and it's usable, but it still has blind spots. To move forward, we need better data, more variety in failure cases, and possibly new models that can capture long-term behavior and edge cases more effectively. These improvements are already on the roadmap for the next phase of this work.

4.5 Next Steps and Future Improvements

Although the system already works and brings value, it's clear that this is only a first version. Several directions are already in mind to make it more complete, more accurate, and better integrated into real workflows.

One of the main improvements would be extending the dataset. As discussed earlier, failures are rare and sometimes inconsistent, which makes it hard for the model to fully learn how to recognize them. Collecting more failure cases (ideally with better annotations and richer context) would make a big difference, also in reducing false negatives.

A second area of improvement concerns the type of data we use. Right now, the model sees each device as a one-time snapshot, disconnected from its actual history. In the real world, though, malfunctions tend to build up over time. Including time-based information (how many hours the device has been in use, how many times it was recalibrated, or when it was last serviced, etc.) could help the model distinguish between a harmless deviation and the early signs of degradation. This would probably require rethinking part of the pipeline and moving

toward models that can handle sequences, such as time-series models or recurrent neural networks.

Another step would be making the model easier to update. The current version is trained once and then exported as a static .pkl file. That works for now, but in a production setting things change: new document formats, hardware revisions, and shifts in failure types are all expected over time. Having a retraining loop would help the system stay accurate. This may be every few months, or triggered when performance drops, for example. Some basic monitoring logic could also be introduced, tracking how the input data shifts and whether the model is still working as expected. These are standard practices in MLOps, and they're worth exploring.

A more technical improvement could involve experimenting with alternative models. Random Forests were a solid choice for this first iteration, especially because they're robust and interpretable. But as the dataset grows, it may be worth trying gradient boosting methods (like XGBoost or LightGBM), or even transformer-based architectures if sequential patterns become important. These models both represent a natural evolution of the random forest, and both could offer a boost in precision and allow the system to pick up subtler patterns.

In short, the foundations are in place. The system can already read technical documents, learn from test values, and identify potential risks. In general, the objective for future improvement is to find a concatenation of modular solutions that would increase our metrics, while still maintaining readability.

Conclusion

This thesis sets out with a practical goal: to explore how AI could help make sense of technical reports, specifically Device History Records (DHRs), and support predictive maintenance in an industrial context. Starting from raw, unstructured documents, the system was built step by step: documents were processed and cleaned, features extracted and structured, a predictive model was trained and evaluated, and finally, the system was integrated into a real-world scenario. What began as a document parsing challenge ended up showing how data, when correctly extracted and interpreted, can actively contribute to decision-making in production environments. The technical achievements of this project are not limited to the model's performance metrics, though these were encouraging: the Random Forest classifier consistently detected about 75% of actual failures in the test set, with limited false positives. More importantly, the system proved to be usable and understandable. Thanks to SHAP-based explainability and the deliberate choice of an interpretable model, predictions could be linked back to concrete test parameters, allowing domain experts to validate and act on them. This transparency is not a side effect, as it is what makes the system trustworthy. One of the most valuable lessons learned was how much value can be recovered from documents that were never meant for machine learning. Technical reports are usually archived and forgotten, often because their structure is inconsistent and their content too variable for traditional analysis. By building a flexible pipeline, based on visual language models and prompt engineering, this project showed that it's possible to turn these noisy inputs into reliable data sources. That data, once cleaned and matched to service outcomes, becomes the foundation for learning useful patterns about device behavior, failure risks, and operational anomalies. Of course, the system is far from perfect: its predictions are probabilistic, not definitive; the data it works on is still limited in size and overall quality; and the class imbalance issue remains a challenge, especially in edge cases. Some failure signals are subtle or context-dependent, and the model trained only on test parameters without external information, can only go so far in detecting them. These limitations are well known and were discussed in detail in Chapter 4, but they do not reduce the value of the system. Instead, they define the path for future work.

Several directions are already clear for extending this project. First, the issue of model drift will become relevant over time: as new devices, hardware revisions, or test protocols are introduced, the model may lose its accuracy. To mitigate this, versioning the model and periodically retraining it with updated data will be essential. Monitoring drift in input distributions and prediction performance could be implemented through existing MLOps tools and should become part of the deployment strategy.

While the model may evolve, and while the pipeline can always be improved, the core idea remains the same: information that was once hidden in static PDFs can now support active decision-making. And that, in itself, marks a meaningful step toward smarter, more efficient maintenance processes.

Appendix A – Code Structure

Throughout this project, several scripts and data files have been developed or processed to build the machine learning pipeline. This appendix provides a brief reference for the most relevant Python scripts and datasets cited in the main text, summarizing their role without duplicating implementation details.

- ***data_cleaning.py***
  Handles the parsing and normalization of raw complaint data. It deals with missing values, formats (dates, numerics), and standardizes textual categories for model input. The resulting structured dataset is saved as ***data_cleaned.xls.***

- ***structural_analysis.py***
  Used during the exploratory phase. This script computes statistical summaries, feature distributions, and correlation matrices. It was instrumental in early feature selection through point-biserial correlation.

- ***ai_model.py***
  The main training script. It loads the cleaned dataset, applies SMOTE to balance classes, trains a Random Forest classifier, and evaluates performance using accuracy, precision, recall, and F1-score. The trained model is exported as random_forest.pkl.

- ***random_forest.pkl***
  A serialized version of the trained model, used for downstream inference or deployment.

- ***data_cleaned.xls***
  The structured dataset resulting from the cleaning process and used as input for training. All features used in the model were selected or derived from this file.

- ***quanta_extraction.xls***
  Contains values automatically extracted from technical documents using prompt-based parsing. This file was used to enrich the training set and to compare model predictions with field outcomes.

- ***database(json)***

  A collection of normalized records in JSON format, used as intermediate storage or for integration into dashboards or APIs.

- ***document_extraction_code***

  A separate set of scripts was developed to extract structured data from visual documents such as test reports or intervention records. These modules include layout recognition logic, prompt templates, and LLM-based parsing through watsonx.ai.

Each file or script was used within a specific phase of the pipeline, from data preparation to modeling and explainability, as shown in Chapter 3.

Appendix B – Prompt Examples

This appendix documents the prompt engineering strategy used to extract and classify information from scanned technical documents. Prompts were passed to a visual language model (Pixtral 12B) through IBM watsonx.ai and served two main purposes: **classifying document types** and **extracting structured tabular data**. All prompts were implemented using the LangChain interface and loaded dynamically depending on the document type.

B.1 Classification Prompt

The script `classify_orders_image.py` handles the classification of individual scanned pages by prompting the model to read specific elements in predefined regions of the page layout: the code in the top-right, the revision letter below it, and the page number in the bottom-right corner.

This is the exact prompt used:

```
Your job is to extract the code that you find in the top right corner of
the image, the revision letter below it and the number of the page in the
bottom right corner.

The code on the top right could be one of the following:
SC_TBO000038
SC_TBO000039
SC_TBE000010
SC_TBX000010
SP_PWMS00048

The revision letter below the code could be one of the following:
Rev_A
Rev_B
Rev_C
Rev_D
Rev_E
Rev_F

YOU MUST ONLY RETURN THE CODE THAT YOU FIND WITH THE PAGE NUMBER IN THIS
FORMAT [CODE, REVISION, PAGE]
IF you don’t find a value simply return none

Output examples:
[SC_TBO000038, REV_A, 1]
[SC_TBO000010, REV_B, 3]
[SC_TBX000010, REV_C, 1]
[none, none, none]
```

```
[SP_PWMS00048, none, 1]
[SC_TBE000010, REV_A, 5]

FOLLOW STRICTLY THE STRUCTURE OUTPUT OF THIS EXAMPLES
ANY OTHER FORMAT RESULT MUST BE IGNORED
```

This prompt is injected into the LangChain interface along with the scanned image encoded in base64. The model's output is parsed and validated to ensure format consistency.

B.2 Extraction Prompt with JSON Template

The second major type of prompt is used for **structured field extraction** from technical tables. This logic is implemented in the script `img_extraction_data.py`, which loads a dynamic prompt from `prompt_template.txt`. A placeholder in the template is replaced at runtime with the correct JSON format required for the specific document type.

Below is the generic form of the prompt:

```
Your duty is to extract the information contained in the table found in the
image and export it into a json_format.

The json_format has for each data entry a comment that explains the
guidelines for each value, these are always true, and you must follow them.

The output MUST BE only the rewrited json_format, you will be penalized for
the text before the json_format and after the json_format.

Blank data or null data should be "null", DON'T make up data.

You must follow this order:
- Put only the data you see on the table in the following json_format.
<REPLACE_THIS>
- Write only the json_format
```

A real example of JSON format passed to the model is:

```
{
  "Verifica potenza in uscita": {
    "short pulse (Master Pulse = 0)": {
      "power output (W)": "float", // must be between 20 and 50
      "t (us)": "int"              // must be between 500 and 700
    }
  },
  "Posizionamento Galvo": {
    "L1": "int", // always 5 digits
    "L2": "int"  // always 5 digits
  }
}
```

This strategy enabled consistent, structured output even across documents with layout variability or partial noise.

B.3 Prompt Execution Logic

Both classification and extraction prompts are embedded in automated chains that:

- Encode the scanned image in base64
- Combine it with the appropriate prompt
- Submit the request to the visual model via watsonx.ai
- Parse and validate the model output using domain-specific rules and regexes

All of this is handled dynamically depending on document type, enabling a fully automated pipeline for handling scanned reports.

B.4 Why Prompt Engineering Matters

In this project, prompts were the only mechanism used to control model behavior—no custom training or fine-tuning was performed. This choice proved effective: thanks to precise and constrained instructions, the model extracted usable data without hallucination and with high consistency.

Prompt engineering acted as the interface between raw documents and structured dataset, making it a core component of the pipeline.

Bibliography

Xu, Y., Li, M., Cui, L., Huang, S., Wei, F., & Zhou, M. (2020). *LayoutLM: Pre-training of Text and Layout for Document Image Understanding*.

Breiman, L. (2001). *Random Forests*. Machine Learning.

Galar, M., Fernandez, A., Barrenechea, E., Bustince, H., & Herrera, F. (2012). *A review on ensembles for the class imbalance problem: bagging-, boosting-, and hybrid-based approaches*. IEEE Transactions on Systems, Man, and Cybernetics.

Choudhary, A., Harding, J., Tiwari, M. K., & Shankar, R. (2009). *Data mining in manufacturing: A review based on the kind of knowledge*. Journal of Intelligent Manufacturing.

Ribeiro, M. T., Singh, S., & Guestrin, C. (2016). *"Why should I trust you?": Explaining the predictions of any classifier*. Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.

Couronné, R., Probst, P., & Boulesteix, (2018). *Random forest versus logistic regression: a large-scale benchmark experiment*. BMC Bioinformatics.

Sewak, M., Sahay, S., & Rathore, H. (2018). *Comparison of Deep Learning and the Classical Machine Learning Algorithm for the Malware Detection*. Procedia Computer Science.

Biau, G., & Scornet, E. (2015). *A Random Forest Guided Tour*.

IBM. (2024). *watsonx.ai – Build AI applications responsibly*.

LangChain. (n.d.). *LangChain Documentation*.

IBM Research. (2024). *Pixtral Multimodal Foundation Model (internal release)*. Retrieved from IBM watsonx.ai documentation.

Web References

https://openai.com/research/gpt-4

https://www.ibm.com/products/watsonx-ai

https://docs.langchain.com/

https://www.mongodb.com/