Tampere University of Applied Sciences

End-to-End Predictive Maintenance Pipeline for Elevator Operations

Xiaosi Huang

BACHELOR'S THESIS
March 2025

Bachelor's Degree Programme in Software Engineering

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Bachelor's Degree Programme in Software Engineering

XIAOSI HUANG:
End-to-End Predictive Maintenance Pipeline for Elevator Operations

Bachelor's thesis 55 pages, appendices 8 pages
March 2025

This thesis develops a scalable predictive maintenance system for elevator operations by integrating gradient-boosted machine learning models, cloud-based automation, and interactive web visualization into a comprehensive end-to-end pipeline. A Gradient Boosted Trees (GBT) classifier was trained using PySpark and MLflow on the Databricks Lakehouse platform. The trained model was then deployed on a cloud-based virtual machine provisioned through the CSC Pouta Infrastructure-as-a-Service (IaaS) environment, enabling automated inference via scheduled execution. The system continuously monitors failure probabilities, triggers alerts based on defined thresholds, and delivers insights through automated email notifications and an interactive web dashboard built with Flask. The key contribution of this project lies in constructing a modular and reproducible predictive maintenance pipeline that demonstrates real-world applicability to industrial equipment monitoring tasks.

Key words: predictive maintenance, machine learning, Databricks Lakehouse, Delta Lake, PySpark, cloud automation, IaaS, Flask web application

CONTENTS

ABBREVIATIONS AND TERMS

ACID	Atomicity, Consistency, Isolation, Durability
AUC	Area Under Curve
API	Application Programming Interface
AWS EMR	Amazon Web Services Elastic MapReduce
BI	Business Intelligence
cPouta	CSC's Cloud Computing Service for Research
CSC	IT Centre for Science (Finland)
CSV	Comma-Separated Values
DBFS	Databricks File System
EDA	Exploratory Data Analysis
FN	False Negative, incorrectly predicted as negative
FP	False Positive, incorrectly predicted as positive
GBT	Gradient Boosted Trees
IaaS	Infrastructure-as-a-Service
IoT	Internet of Things
JVM	Java Virtual Machine
LSTM	Long Short-Term Memory
ML	Machine Learning
NaN	Not a Number
PdM	Predictive Maintenance
RAM	Random Access Memory
ROC	Receiver Operating Characteristic
RUL	Remaining Useful Life
SQL	Structured Query Language
TAMK	Tampere University of Applied Sciences
TCP	Transmission Control Protocol
TN	True Negative, correctly predicted negative class
TP	True Positive, correctly predicted positive class
UDF	User-Defined Function
Web UI	Web-based User Interface
WSGI	Web Server Gateway Interface

1 INTRODUCTION

Predictive maintenance is a cutting-edge strategy that uses sensor data and data analysis technology to monitor equipment status in real time, thereby identifying potential problems before failures occur. Compared with traditional preventive maintenance, predictive maintenance relies on real data from equipment operation rather than fixed maintenance cycles based on statistical lifespans. Therefore, it can arrange maintenance tasks more flexibly and reduce the risk of unplanned downtime (KONE, 2025).

As an important part of urban infrastructure, elevator systems are highly sensitive to mechanical wear, ambient humidity, vibration and other factors in high-frequency, high-load operating environments. In terms of public safety and continuity of service, any failure can have serious consequences. It has therefore become a critical need to detect and intervene in advance of abnormal conditions in key elevator components. Through real-time monitoring and data modelling, predictive maintenance can provide more predictive decision support for elevator operation and maintenance.

The "24/7 Connected Services" predictive maintenance platform launched by KONE and data analysis company Solita demonstrates the powerful potential of modern analytical technology in this field. The platform collects equipment status data in real time based on IoT sensors, then combines cloud analysis with machine learning models to analyse and model abnormal fluctuations in elevator operation. Based on available case data, the deployment of the platform was associated with a 40% reduction in maintenance requests, a 50% decrease in passenger entrapment incidents, and early identification of approximately 70% of faults (Solita, 2025).

The strength of such predictive systems lies in the ability to establish a data-driven loop that spans sensor data collection, model-based risk analysis, automated alerts, and visualization feedback. This thesis aims to design and implement a fully integrated predictive maintenance system for elevator equipment.

The system covers the complete pipeline from data ingestion, cleaning, and feature engineering to model training, remote deployment, scheduled execution, and real-time alerting. Furthermore, it provides a web-based interface to visualize equipment health status. With GBTClassifier as the central prediction model, and utilizing technologies such as the Databricks Lakehouse architecture, MLflow for experiment tracking, CSC Pouta for cloud deployment, and Flask for web visualization, this project delivers a practical, reproducible, and scalable solution for intelligent maintenance with broader potential for industrial applications.

In conclusion, the application of predictive maintenance in elevator systems is becoming increasingly feasible with the advancement of IoT and AI technologies. By building an end-to-end system with automation and visualization capabilities, this project aims to enhance equipment reliability and operational safety while contributing to the digital transformation of urban infrastructure maintenance.

2 PROJECT BACKGROUND AND KEY TECHNOLOGIES

2.1 Project Objective and Scenario

This project aims to develop a predictive maintenance system for elevator operations, capable of providing end-to-end automation from sensor data to failure warning. The system utilizes sensor data collected through IoT, and a supervised machine learning model is trained to predict potential failures in elevator door mechanisms. Data processing and model development are conducted on the Databricks Lakehouse platform, which provides scalable computation and version-controlled support for the entire pipeline.

The architecture includes data processing, model training, automated deployment, and visualization. All automated through scheduled tasks on a CSC Pouta cloud server, a cloud-based IaaS environment, and is capable of hourly automated inference and remote alerting.

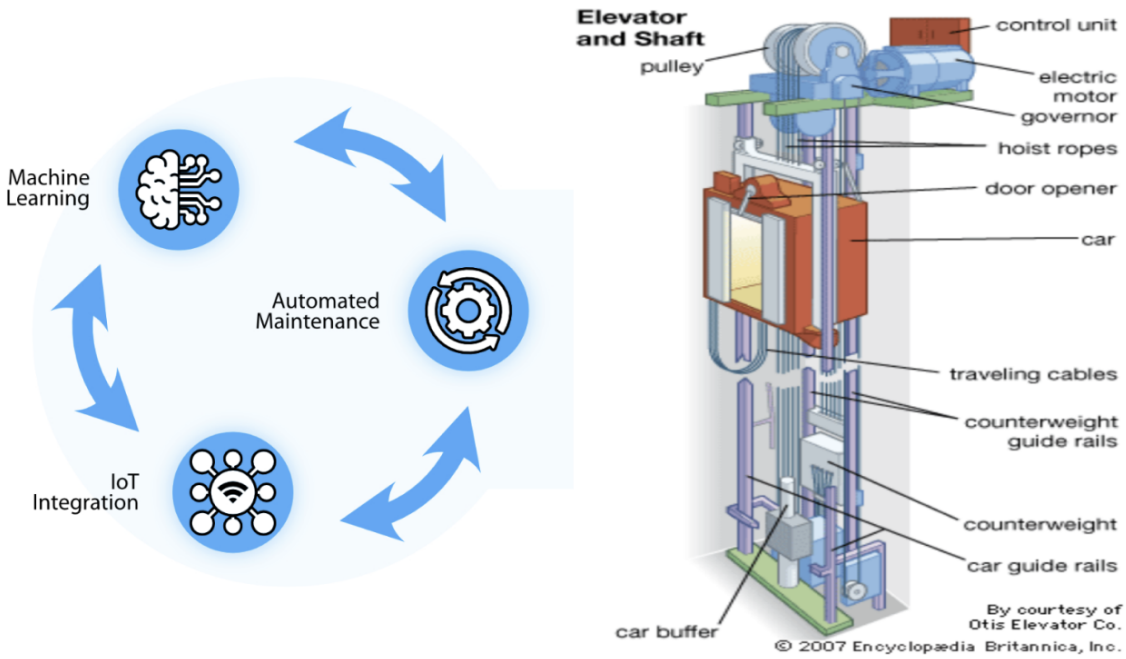
By combining IoT sensing, machine learning, and web-based reporting, this solution provides a scalable framework for predictive maintenance in industrial environments where safety and uptime are critical.

2.2 Predictive Maintenance in Elevator Industry

Predictive maintenance is a proactive strategy that utilizes sensor data and analytical models to detect early signs of equipment degradation before actual failures occur. By continuously monitoring the operational status of mechanical systems and analysing patterns over time, it enables timely interventions that can significantly reduce unexpected downtime and maintenance costs (Mobley, 2002).

In the elevator industry, where equipment reliability and passenger safety are critical, traditional reactive or periodic maintenance strategies are no longer sufficient. Sudden failures in elevator doors can lead to service disruptions, safety

hazards, and customer dissatisfaction. Vibration monitoring has emerged as a core enabler of predictive maintenance in elevator systems. By continuously recording motor vibration patterns and analysing deviations from standard behaviour, maintenance teams can detect mechanical issues.



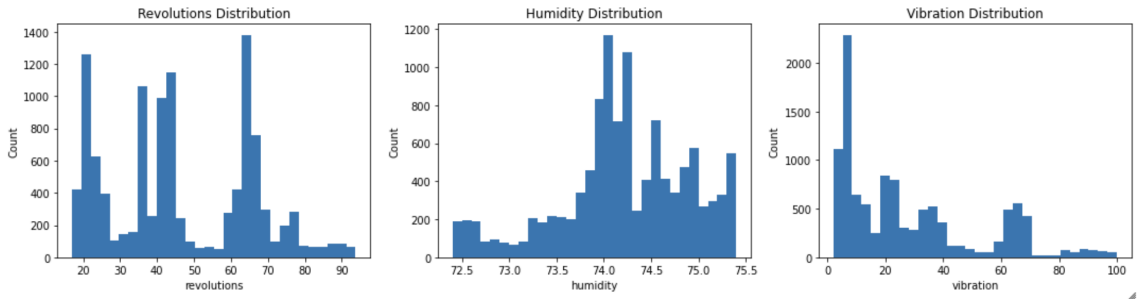
Picture 1. Predictive Maintenance Framework in Elevator Systems (Yordanov & Britannica, Predictive Maintenance in Lift Systems & Elevator Components, 2023; 2007)

As illustrated in PICTURE 1, modern predictive maintenance frameworks integrate IoT sensors, machine learning models, and real-time automated responses to deliver a complete monitoring and alerting cycle. This approach not only reduces safety risks but also enables data-driven decision-making for long-term reliability and cost efficiency in elevator operations (Yordanov, Lift Motor Monitoring and Real time Failure Prevention Predictive Maintenance of Lift Installations, 2023).

2.3 Dataset and Development Environment

The dataset used in this project is the Elevator Predictive Maintenance Dataset, which is provided by Huawei German Research Centre and publicly released on the Kaggle platform. This dataset is collected from various IoT sensors of the

elevator door opening and closing system, covering key parameters such as motor rotation speed, environmental humidity, and physical vibration. It is mainly used to evaluate and model the health status of elevator door components in predictive maintenance scenarios.



Picture 2. Histograms of Revolutions, Humidity, and Vibration

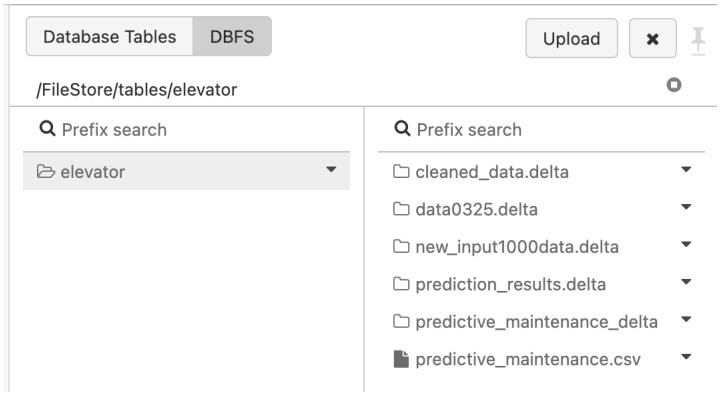
The data is recorded in a sequence with a frequency of 4Hz, covering the daily peak and night operation period from $16:30$ to $23:30$. The raw data is provided in CSV format, with the file name $\text{predictive\_maintenance.csv}$, containing a total of $112{,}001$ rows and 9 columns. All the variables in the table are numerical. The target variable is Vibration, and the other parameters related to the device are Revolutions, the humidity of the environment, and 5 anonymous sensors $\text{x1}-\text{x5}$.

```
Dataset contains 112001 rows and 9 columns.
+------+-----------+--------+---------+------+------+------+------+------+
|    ID|revolutions|humidity|vibration|    x1|    x2|    x3|    x4|    x5|
+------+-----------+--------+---------+------+------+------+------+------+
|112001|     112001|  112001|   109563|112001|112001|112001|112001|112001|
+------+-----------+--------+---------+------+------+------+------+------+
```

Picture 3. Raw Data Summary and Missing Value in the Vibration Column

Model development and experimentation were created on the Databricks Lakehouse platform, a cloud-native environment built on Apache Spark. This platform offers scalable computation, integrated notebook support, and built-in version control. In the development process, raw CSV data were uploaded to the Databricks File System (DBFS). The Spark runtime automatically inferred column types and supported downstream operations like missing value handling, feature engineering, and model training.

Faced with limited computing resources on the Databricks Community Edition, including limited RAM and a single-node runtime, the project applied data sampling and streamlined transformation steps. Although the platform has these limitations, it still supports a full pipeline from data ingestion to model export, ensuring reproducibility and consistency throughout the process.



Picture 4. Data File Storage Structure in the Databricks Lakehouse Platform

In addition, the project integrates several essential Python libraries to support analytics and visualization tasks. PySpark was used for data preprocessing and model training, while MLflow handled the tracking of experiments and model versions. For local data manipulation, the project relied on Pandas and NumPy, and exploratory data analysis was conducted using Matplotlib and Seaborn. Evaluation metrics and ROC curve generation were carried out with the help of scikit-learn. All these components were seamlessly coordinated within the Databricks notebook environment, enabling a modular, maintainable, and fully end-to-end machine learning workflow.

2.4 Data Platform and Architecture

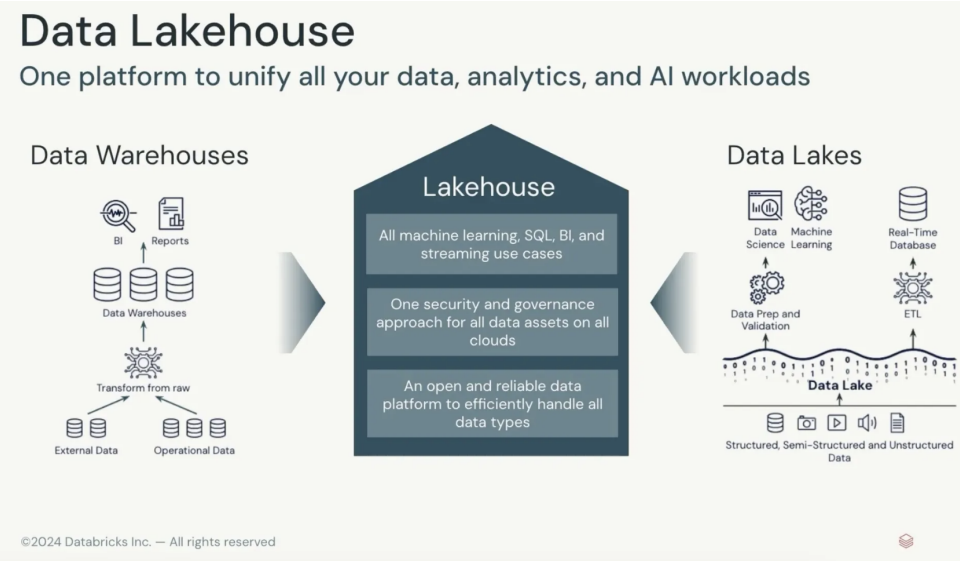
Predictive maintenance systems involve a large amount of high-frequency data from IoT sensors, requiring a platform that not only offers efficient processing capabilities but also supports modelling, analysis, and visualisation. The Lakehouse model combines the flexibility of a data lake with the transactional consistency of a data warehouse, enabling the full data collection process, storage, analysis, modelling and deployment (Michael Armbrust, 2020). This architecture is particularly suitable for heterogeneous data processing and machine learning

training tasks in industrial scenarios, significantly reducing engineering complexity and latency associated with traditional multi-platform setups.

2.4.1 Databricks Lakehouse

Databricks Lakehouse is a unified data architecture that combines the elasticity of a data lake with the strong transactional support of a data warehouse. In traditional architectures, data lakes are good at storing massive amounts of unstructured data, but do not support transactional consistency or optimized query performance; while data warehouses are suitable for structured data analysis, they are often limited in scalability and flexibility compared to modern architectures. Lakehouse uses Delta Lake technology to overlay a transactional metadata management mechanism on top of the data lake, implementing advanced features such as ACID transactions and Schema evolution (Armbrust, et al., 2020).

Lakehouse supports efficient reading and updating of sensor data, allowing feature generation, label construction, and iterative training in the model training process to be performed under a unified data view. Its powerful version control also ensures data consistency between different model experiments, thereby significantly improving the stability and tuning efficiency of the prediction model.

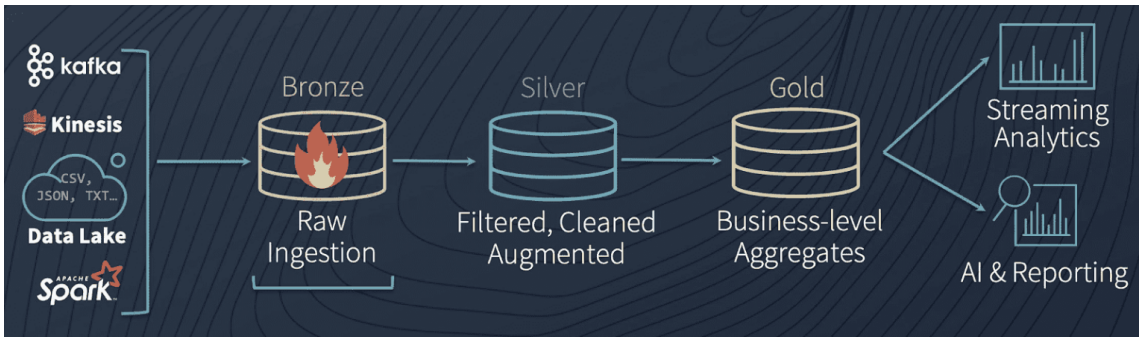


Picture 5. Lakehouse Architecture: Unifying Data Lakes and Data Warehouses (Databricks, 2024)

Lakehouse combines the advantages of data warehouse and data lake in terms of function, as shown in PICTURE 5, on the left, the traditional data warehouse supports BI reports and SQL queries for structured data; the data lake on the right is suitable for large-scale unstructured data, stream processing and machine learning. Lakehouse, as an intermediate fusion platform, supports unified metadata governance, security control and cross-platform analysis, and improves the consistency of modelling and the transparency of data processing in elevator predictive maintenance tasks.

2.4.2 Data Architecture

In the Databricks platform, there are two main ways to store data: Databricks File System and Delta Table. In the early stages of this project, the raw sensor data was uploaded to the path /FileStore/tables/ in DBFS in CSV file format and read in the form of Spark DataFrame. This method has the characteristics of low entry threshold and flexible reading, which is suitable for prototype development and initial modeling. However, as data cleaning and model iteration go deeper, the demand for transaction consistency, version control, and data tracking gradually increases. At this point, Delta Table provides more powerful support. It is built on Apache Parquet and enhances the ACID transaction features, supporting time travel, automatic schema evolution, and fine-grained data governance capabilities (Brenner Heintz, 2019).



Picture 6. Delta Lake Bronze–Silver–Gold Architecture
(Databricks, Productionizing Machine Learning with Delta Lake, 2019)

Delta Table organizes data in a three-layer structure of "Bronze-Silver-Gold": the bronze layer is responsible for the access and persistence of raw data, the silver

layer is responsible for filtering, cleaning and replenishment, and the gold layer is used for aggregation, modelling and visual analysis. This structure makes the data engineering process clearer and greatly improves the stability of model training and online prediction.

In terms of resource control, the community version of Databricks has restrictions on DBFS storage space and does not support automatic version control, while Delta Table has stronger fault tolerance and concurrency performance in the enterprise version. DBFS is initially used for development and debugging, and the modelling and prediction processes are migrated to the Delta table structure to achieve more stable maintainability and engineering standardization.

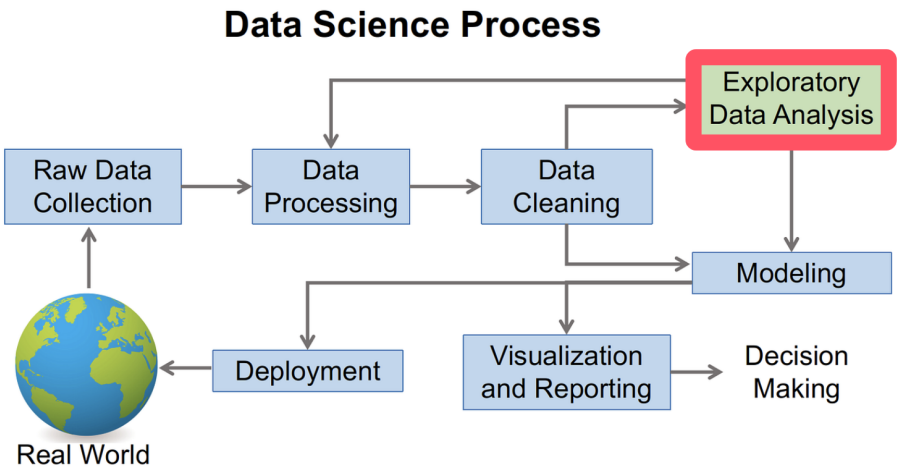
2.5 Modelling Concepts and Design

The supervised learning workflow covered all key steps needed for predictive modelling, including feature construction, model selection, training control, and performance evaluation. A Gradient-Boosted Trees (GBT) classifier was chosen as the main model; it can handle complex feature sets and maintain strong generalization. To help the model recognize patterns in time-series data more effectively, lag features were created using a sliding window technique. This approach allowed the model to learn from recent historical trends in sensor readings, making it more responsive to gradual changes in equipment condition.

2.5.1 EDA and Feature Engineering Concepts

In machine learning projects, EDA (Exploratory Data Analysis) is a key pre-step in the modelling process, mainly used to understand data distribution, discover potential relationships between variables, and identify missing values and outliers (malamahadevan, 2025). In this project, EDA was applied to the visual analysis of elevator sensor data to help identify the main variables that affect equipment performance, such as the obvious upward trend of vibration before equipment

failure. By drawing histograms and correlation heat maps, it was preliminarily determined that there was a strong negative correlation between vibration and remaining life. This finding directly determined the solution for feature construction.



Picture 7. Exploratory Data Analysis: A Critical Step Before Modelling (MarkovML, 2024)

Exploratory data analysis serves as a bridge between raw data collection and meaningful inference. Within this project, EDA played a similar role by uncovering early signals of degradation through statistical visualization, helping to identify the strong correlation between vibration levels and the remaining useful life (RUL) of elevator components.

Lag features refer to using variable values at several moments in the past as inputs at the current moment. Feature engineering involves creating new features from the existing data to enhance the model (Baig, Canamusa, Leskinen, Happonen, & Leppänen, 2025). For instance, the current vibration value may reflect not just the present state of the system but also be influenced by changes that occurred in the previous 3 to 5 seconds. By constructing variables such as vibration_lag_1 and vibration_lag_2, the model can learn and capture the dynamic changes in the equipment state, which can more accurately predict future trends.

In International Standards about condition monitoring and diagnostics of machines, RUL is defined as remaining time before system health falls below a defined failure threshold, or before the system passes into a state in which it needs to be repaired or replaced (Katser, 2023). On the other hand, the project uses

Remaining useful life as one of the core indicators for fault prediction. RUL represents the available time between the current moment and the complete failure of the equipment. Each time series sample is counted backward until its corresponding vibration exceeds a set threshold such as 1.0, and then the remaining number of cycles is calculated as the RUL value. Creating RUL enhances the model's sense of temporal dynamics and enables the transformation of binary fault labels into continuous regression targets.

Through the above feature engineering process, the original elevator sensor data is effectively structured and converted into a high-dimensional input tensor containing historical dynamics and fault trends, which provides effective input information for GBTClassifier model training.

2.5.2 Implementation of GBTClassifier Model in PySpark

Gradient Boosted Trees (GBT) is an ensemble learning method based on an additive model. It gradually approximates the target function by serially combining multiple weak learners, usually shallow decision trees. It is one of the most widely used nonlinear modelling techniques in industrial predictive maintenance (Chen, Huang, Cohn, Zhang, & Zhou, 2022). Compared with traditional single decision trees, GBT can better handle complex interactions between features and has stronger robustness when facing high-dimensional and noisy data. Therefore, it is particularly suitable for device status prediction tasks in IoT scenarios.

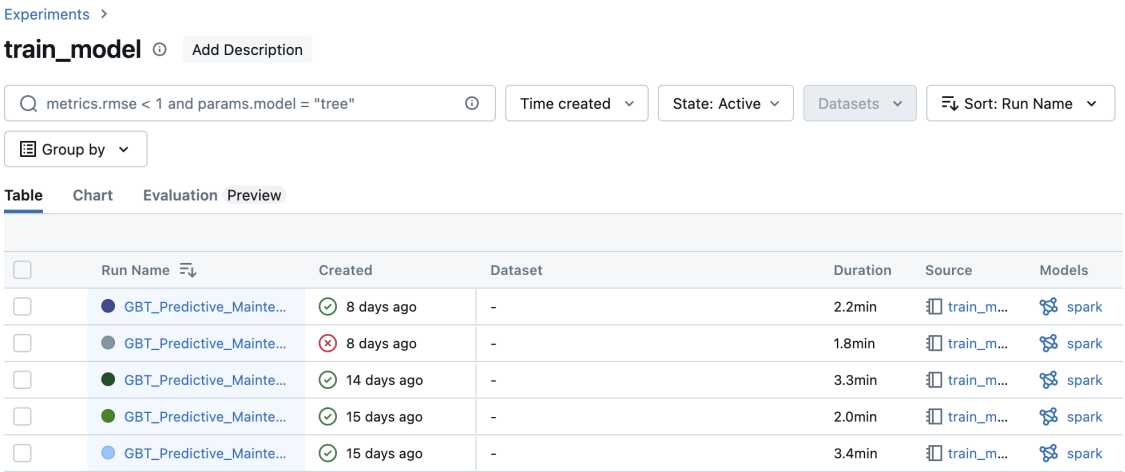
The project utilises GBTClassifier from the $\mathrm{pyspark.ml.classification}$ module, executed on the Databricks-based PySpark framework. The model improves its predictive accuracy by successively fitting new decision trees to correct the residuals of prior iterations. In the specific implementation process, sensor data under multiple historical time windows are integrated into feature vectors and standardized to enhance the convergence efficiency of the model. Finally, the binary classification modelling of "fault" and "non-fault" states is completed in combination with GBTClassifier

The final model evaluation used AUC and ROC curve indicators. The results showed that the GBT model has good predictive ability in identifying potential elevator door failures, verifying its application value in predictive maintenance (Databricks, pyspark.ml.classification.GBTClassifier, 2025).

2.5.3 MLflow for Experiment Tracking

In the process of machine learning modelling, experimental management is to ensure model repeatability and result traceability. MLflow is an open-source machine learning lifecycle platform designed for managing experimental processes. It supports parameter recording, result indicator tracking, model version control, and visual management (Databricks, 2025).

During the development of the elevator predictive maintenance system, MLflow was integrated into the Databricks platform to automatically record each GBTClassifier training process, including key information such as input feature dimensions, training set ratio, AUC value, and recall rate. All experimental data is uniformly stored in the Databricks experiment page and can be viewed intuitively through the Web UI.



Picture 8. MLflow Tracking of GBTClassifier Training Runs

This MLflow interface on Databricks Community Edition clearly records each run of the GBTClassifier model training process, including feature dimensions, evaluation metrics, and run durations. The final model persisted and exported via

MLflow's model management and successfully deployed to the remote Pouta cloud platform.

2.5.4 Evaluation Metrics: AUC, ROC, Confusion Matrix

To evaluate the performance of the GBTClassifier model in predicting whether an elevator equipment is about to fail, a variety of classification evaluation indicators are used. The most critical ones include AUC (Area Under Curve), ROC (Receiver Operating Characteristic) curve and confusion matrix.

AUC represents the area under the ROC curve, which is used to measure the model's ability to distinguish between positive and negative samples at different thresholds. The closer the AUC is to 1, the better the performance (Fawcett, 2006).

The confusion matrix shows the four prediction results of TP, TN, FP, and FN, which can be further calculated for accuracy, recall, and precision. Considering the imbalance of data, recall is particularly important to reduce the risk of missing faults. (Takaya Saito, 2015)

(1)

$$Accuracy = (TP + TN) / (TP + TN + FP + FN)$$
$$Precision = TP / (TP + FP)$$
$$Recall = TP / (TP + FN)$$

Where:

TP (True Positive): Number of correctly predicted positive instances
TN (True Negative): Number of correctly predicted negative instances
FP (False Positive): Number of negative instances incorrectly predicted as positive
FN (False Negative): Number of positive instances incorrectly predicted as negative

These metrics provide a comprehensive evaluation of the model's classification performance. And these indicators also ensure the GBTClassifier is both effective and reliable for predictive maintenance applications.

2.6 Cloud Platform and Scheduling

Cloud platforms and scheduling tools play a critical role in operationalizing machine learning models. This section presents the cloud deployment strategy using CSC Pouta and explains how Cron is used to schedule regular inference jobs in prediction task scheduling.

2.6.1 CSC Pouta and IaaS

This project uses CSC Pouta, the Finnish national scientific cloud computing platform, which is built on OpenStack and provides elastic cloud resources in the Infrastructure as a Service (IaaS) model. Users can create virtual machine instances according to project requirements, configure CPU, memory, and storage resources, and remotely manage the model deployment environment through SSH.

Pouta's community cloud cPouta is particularly suitable for academic research, with good security isolation and resource visualization management interface. It supports data volume mounting, persistent storage and public network access, which greatly facilitates the transfer process of machine learning models from local training to online deployment (CSC – IT Center for Science, 2025).

With IaaS mode, users do not need to worry about the maintenance of the underlying hardware but only need to use computing and storage resources on demand, with good scalability and deployment flexibility (Jyväskylä University of Applied Sciences, 2025). In this project, the trained GBTClassifier model is encapsulated and uploaded to the Pouta instance to achieve remote real-time inference.

2.6.2 Cron and Cron Job Scheduling

In cloud deployment, the model's inference tasks need to be executed regularly at a fixed frequency to achieve continuity and automation of predictive maintenance. To this end, the system uses Cron, a classic task scheduling tool in the

Linux operating system, to manage the periodically running inference script predict_with_model.py.

Cron is a built-in job scheduler commonly used in Unix-like operating systems, designed to execute predefined scripts or commands automatically based on a fixed timetable. Its scheduling mechanism is controlled through the crontab configuration file, where time expressions follow a “minute-hour-day-month-weekday” syntax. This format enables flexible scheduling options, allowing tasks to run on an hourly, daily, weekly, or monthly basis (Ubuntu Community, 2016).

Within the current setup, the inference script is triggered precisely at the start of every hour. The execution output is appended to a local log file, while a built-in mail notification system forwards the job status to a designated email address. This integration not only improves system traceability but also enhances long-term maintainability in unattended environments.

2.7 Web Interface and Visualization

To enhance the usability and responsiveness of the predictive maintenance system, this project developed a browser-accessible web interface that displays model inference results in the form of charts and interactive tables. The interface enables real-time status feedback for maintenance teams, transforming raw prediction outputs into actionable insights.

2.7.1 Flask as the Backend Framework

As a backend development framework, Flask is a micro web framework written in Python. It has the advantages of flexible development, clear structure, few dependencies, and easy deployment. It is suitable for quickly building lightweight data service interfaces (Grinberg, 2018).In this project, Flask is used to build the model output interface and front-end rendering logic. By encapsulating the API interface, the backend can return the model inference results in JSON format and pass it to the front-end page for dynamic rendering.

In addition, the template engine Jinja2 provided by Flask supports HTML pages to embed dynamic variables, and combined with JavaScript plug-ins such as DataTables, it can realize interactive functions such as data paging, sorting, and keyword search. The project uses the Flask Blueprint module to organize various functional routes, and divides different page functions into independent sub-modules, such as forecast table view, trend analysis view, and high-risk warning view to achieve clear layering and easy scalability of code logic.

2.7.2 Development with Gunicorn

Gunicorn, as a WSGI (Web Server Gateway Interface) compatible web application server, wraps the Flask application as a sustainable backend service. Gunicorn was adopted as the WSGI-compliant production server. Gunicorn is a Python HTTP server known for its efficiency and ability to handle concurrent requests via multi-process architecture (Gunicorn Developers, 2025). TCP port 5000 is pre-opened in the server security group to ensure that the front-end page can access the service through the public network address.

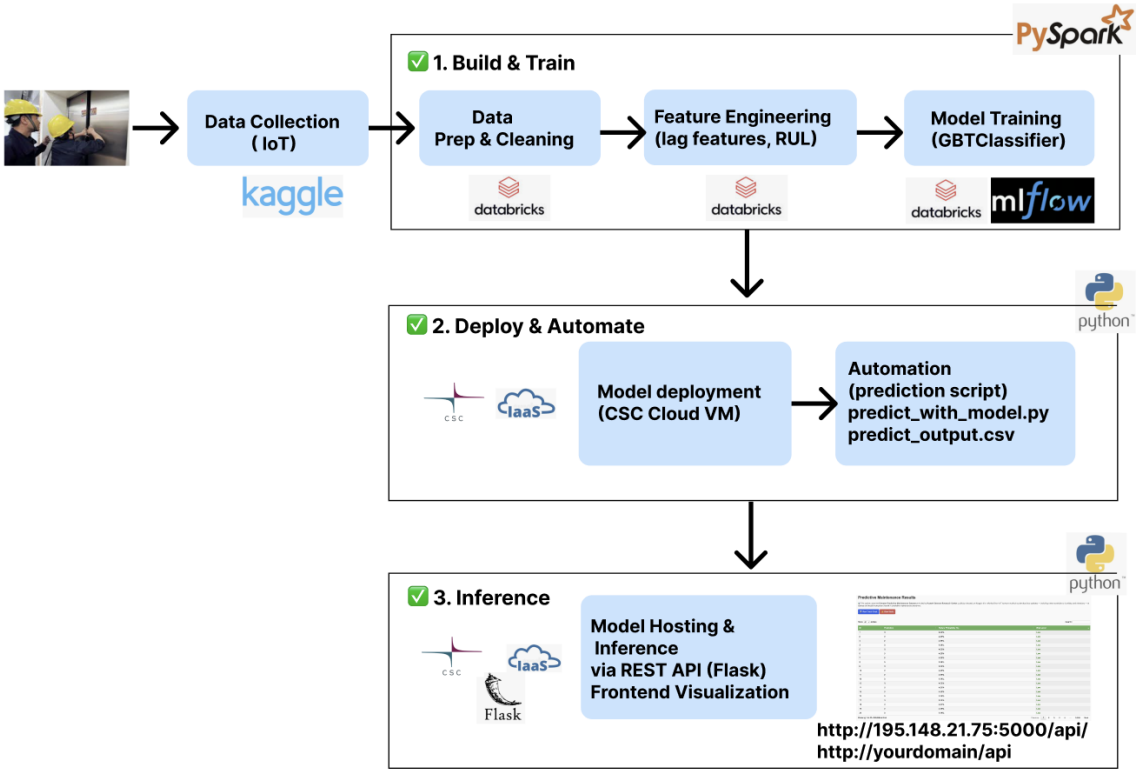
By combining Flask for rapid backend development and Gunicorn for production-grade deployment, the system achieves both ease of implementation and operational robustness, delivering a responsive and maintainable web visualization module that supports real-time predictive maintenance feedback.

This chapter outlines the main design goals of the project, introduces the dataset, and details the entire modelling process. It also provided a clear overview of the technical platforms and system architecture involved.

3 SYSTEM IMPLEMENTATION

3.1 End-to-End System Workflow

This section provides a detailed overview of the end-to-end system workflow developed for predictive maintenance of elevator operations. The solution integrates multiple critical stages including data preparation, feature engineering, model training, cloud deployment, automation, and real-time inference, to form an end-to-end solution.



Picture 9. End-to-End Predictive Maintenance System Workflow: from IoT-based data collection and model training to cloud deployment, automation, and real-time API-based inference.

As illustrated above, this modular and technology-integrated design ensures scalability and reproducibility while enabling a seamless transition from IoT sensor data collection to actionable maintenance insights. The following subsections will describe each major phase in detail, starting with data cleaning and preprocessing on the Databricks platform.

3.2 Data Cleaning and Preprocessing on Databricks

The raw data generated from IoT devices is loaded into the Databricks Lakehouse platform, where initial data integrity checks and preliminary cleaning operations are performed. An analysis of the raw dataset revealed that the vibration column contained a considerable number of missing values (NaN), totally $2{,}438$ rows. Given that vibration is the key target variable in the prediction task, imputing or interpolating missing values may lead to statistical bias and affect the reliability of the model outcomes. Therefore, this project deleted the rows containing missing values. The original raw data set was cleaned using Spark's $\mathrm{dropna}()$ method, and the data volume was reduced from the initial $112{,}001$ rows to $109{,}563$ rows, and a total of $2{,}438$ rows with missing values were removed.

```
Number of missing values in 'vibration': 2438
Original row count: 112001
Cleaned row count: 109563
Number of rows removed: 2438
+---+-----------+--------+---------+---+---+---+---+---+
| ID|revolutions|humidity|vibration| x1| x2| x3| x4| x5|
+---+-----------+--------+---------+---+---+---+---+---+
|  0|          0|       0|        0|  0|  0|  0|  0|  0|
+---+-----------+--------+---------+---+---+---+---+---+
```

Picture 10. Data Cleaning Result After Removing Missing Vibration Values

In terms of feature selection, this project retains the ID as the sequence index and selects revolutions, humidity, and vibration as input feature columns. The time series sliding window features will be constructed based on these three later. To generate the label variables required by the model, the project uses the concept of "Remaining Useful Life" (RUL). The global maximum value of vibration is obtained through the window function, and based on this, the RUL of the current record is calculated:

$$\text{RUL} = \max(\text{vibration}) - \text{vibration}. \tag{2}$$

To convert the continuous RUL into a binary label, the threshold is set to 30. When the RUL of a record is less than or equal to 30, it is marked as a potential fault ($\text{label} = 1$), otherwise, it is marked as a normal state ($\text{label} = 0$).

To further advance the modelling time series, the project adopts a sliding window strategy to construct lag features with a length of 40. For each input variable, a delayed version of the past $40 - \text{time}$ steps is generated, forming a total of 120 feature columns. Since the sliding window operation will cause the previous part of the sample to lose necessary historical data, resulting in missing values in the newly generated feature column, the $\text{dropna}()$ method is used again to clear these incomplete sample records.

After completing the construction of the lagged features, the project uses VectorAssembler to merge all lagged variables into a column of feature vectors assembled_features to meet the model's input requirements for a unified feature structure. The feature vector is standardized through StandardScaler to generate an input vector features with a unified numerical scale for model training. This process enhances the model's ability to capture time series trends and improves the numerical stability of feature expression.

3.3 Model Training and Evaluation

After completing the feature engineering process, a binary classification model was built using the GBTClassifier from PySpark's MLlib. The goal of the model is to predict whether a given equipment state indicates the need for maintenance ($\text{label} = 1$). The modeling process was encapsulated using a Pipeline structure, which consisted of three main stages: feature vector assembly via VectorAssembler, feature standardization using StandardScaler. To evaluate the model's generalization ability, the dataset was randomly split into training and testing sets in a $9{:}1$ ratio. This approach ensures that the model is trained on most of the data while maintaining an independent subset for unbiased performance evaluation.

To enable complete traceability of the training process and reproducibility of results, the project integrated MLflow to track key elements such as model hyperparameters, performance metrics, and model artifacts. After training, the model had persisted in the native Spark ML format and stored at the specified path $/\text{FileStore}/\text{models}/\text{gbt\_pipeline\_model}/$. This storage format preserves the entire pipeline structure, including preprocessing steps and the trained GBTClassifier,

ensuring that the model can be reloaded and reused without loss of configuration. The model was successfully reloaded using $\mathrm{PipelineModel.load()}$, confirming the integrity of the saved artifact and validating the effectiveness of the persistence and restoration process. This workflow allows for seamless model versioning, auditing, and future deployment in production environments.

```
from pyspark.ml import PipelineModel

# Load the saved model
loaded_model = PipelineModel.load("dbfs:/FileStore/models/gbt_pipeline_model")

# Use the loaded model to make predictions again
predictions = loaded_model.transform(test_data)
predictions.select("prediction", "probability").show(5)
```

▸ (16) Spark Jobs

▸ predictions: pyspark.sql.dataframe.DataFrame

```
+----------+--------------------+
|prediction|         probability|
+----------+--------------------+
|       0.0|[0.95635347857270...|
|       0.0|[0.95635347857270...|
|       0.0|[0.95635347857270...|
|       0.0|[0.95635347857270...|
|       0.0|[0.95635347857270...|
+----------+--------------------+
only showing top 5 rows
```

Picture 11. Reloading the Trained Model and Prediction Probabilities

The model demonstrated strong predictive performance on the test set, which consisted of $11{,}072$ samples. And $10{,}506$ samples were predicted as "no maintenance required" ($\mathrm{label} = 0$), while 566 samples were predicted as "maintenance required" ($\mathrm{label} = 1$).

A detailed analysis of the confusion matrix reveals that the model correctly identified 562 positive samples and $10{,}505$ negative samples, with only 4 false positives and 1 false negative. These results indicate that the model achieves both high precision and high recall, particularly in detecting maintenance-required cases.

In addition, the model achieved an Area Under the ROC Curve (AUC) value close to 0.99, as evaluated using the BinaryClassificationEvaluator. This suggests that the classifier possesses a high discriminative ability between the two classes.

The low rate of misclassification further confirms the model's robustness and suitability for real-world predictive maintenance applications, where minimizing both false alarms and missed detections is critical.

```
# Distribution of prediction and label
predictions.groupBy("label", "prediction").count().orderBy("label", "prediction").show()
```

▸ (2) Spark Jobs

```
+-----+----------+-----+
|label|prediction|count|
+-----+----------+-----+
|    0|       0.0|10505|
|    0|       1.0|    4|
|    1|       0.0|    1|
|    1|       1.0|  562|
+-----+----------+-----+
```

Picture 12. Prediction Result Distribution

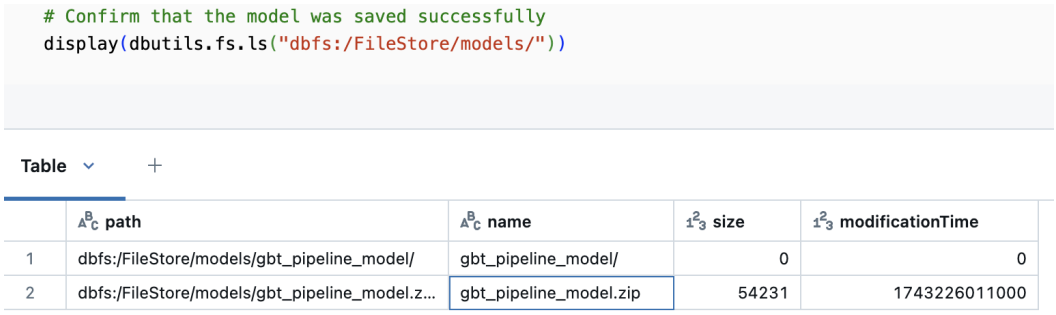
The model showed very strong performance in the elevator fault prediction task. It reached high accuracy and recall while keeping the false alarm rate low. The model correctly identified almost all the cases that required maintenance, with only a very small number of false negatives. This makes the model both reliable and useful in real-world applications, especially in scenarios where missing a fault could lead to safety issues. Its good balance between correct predictions and low false alerts also suggests that it's a strong option for deployment in practical maintenance systems.

3.3.1 Model Export and Migration to CSC Pouta

After completing model training and validation, the trained GBT pipeline model was exported and compressed for reuse and deployment on the CSC Pouta cloud platform. Due to path restrictions in the Databricks Community Edition, the model could not be directly downloaded from the default DBFS location. To work around this, the model was first copied to a temporary local directory `/tmp`, compressed into a `.zip` file, and then uploaded to a DBFS path that supports browser-based access. This made it possible to download the model externally and prepare it for cross-platform deployment.

The trained Spark ML pipeline model was first saved locally at /tmp/gbt_pipeline_model_local using mlflow.spark.save_model() as part of the export workflow. After the model was saved, the entire directory was compressed into a .zip file with the help of Python's shutil module. A compressed archive was created using the shutil.make_archive() command.

Once the .zip file was generated, it was transferred from the local file system back into the Databricks environment using the dbutils.fs.cp() command. The destination path was set to /FileStore/models/gbt_pipeline_model.zip, which supports access through a web browser. The file in a public directory, it became available for external download or transfer to the target deployment platform.



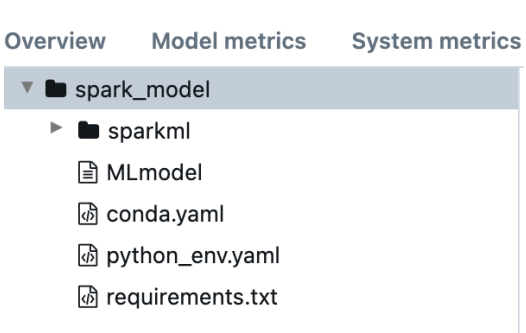
```
# Confirm that the model was saved successfully
display(dbutils.fs.ls("dbfs:/FileStore/models/"))
```

Table +

	path	name	size	modificationTime
1	dbfs:/FileStore/models/gbt_pipeline_model/	gbt_pipeline_model/	0	0
2	dbfs:/FileStore/models/gbt_pipeline_model.z...	gbt_pipeline_model.zip	54231	1743226011000

Picture 13. Saved Model Directory and Compressed Archive in DBFS

To ensure compatibility across different runtime environments, the model export directory includes a set of metadata files automatically generated during the saving process. These files consist of MLmodel, conda.yaml, python_env.yaml, and requirements.txt, each describing the package dependencies, environment configurations, and entry points required for model loading.



Picture 14. Exported Spark ML Model Structure Generated by MLflow

After uploading the compressed model, the .zip file was downloaded and extracted on the target platform. The saved folder included all necessary files and environment info, so the model could be restored directly using $\mathrm{PipelineModel.load()}$ without manually reinstalling any dependencies.

The process confirmed that the Spark ML pipeline could run smoothly outside the original development environment. The files generated by Databricks MLflow, including dependency records and entry points, ensured that the model could be reused without compatibility issues during transfer.

3.3.2 Environment Setup on CSC Pouta

To deploy and run the Spark-based prediction model on the Pouta cloud platform, the required operating environment was manually configured in an $\mathrm{Ubuntu\ 22.04}$ virtual machine instance, which was named $model - vm$ for clarity. After the basic setup, a floating public IP address $195.148.21.75$ was assigned to the instance, and port 22 was opened in the security group settings to allow SSH access from outside the network. Since the default image did not include Java or Spark, the environment had to be configured from scratch to support the loading and execution of the PySpark model.

Spark is a distributed computing framework built on Java. Its operation depends on the Java Virtual Machine (JVM). The first step is to install the Java Development Kit. Use the commands $sudo\ apt\ update$ and $sudo\ apt$ *install openjdk-11-jdk -y* to install the Java Development Kit to provide a basic operating environment for Spark.

A stable version of Spark compatible with $\mathrm{Hadoop\ 3}$ was selected from the official Apache Spark website. In this project, the version $spark - 3.5.5 - bin - hadoop3$ was used. The corresponding $.tgz$ installation package was uploaded to the home directory of the virtual machine using the scp command. Once uploaded, the package was decompressed using tar $-xvzf$ to obtain the complete Spark installation directory.

To simplify access and standardize execution paths, the extracted Spark directory was relocated to $/opt/spark$. $\sim/.bashrc$ file was then modified to include the $SPARK\_HOME$ and $JAVA\_HOME$ variables, along with an updated PATH. After applying the changes with $\text{source} \sim/.bashrc$, the Spark CLI tools became accessible from the terminal.

```
ubuntu@model-vm:/$
ubuntu@model-vm:/$
ubuntu@model-vm:/$ cat ~/.bashrc | grep -E 'SPARK_HOME|JAVA_HOME|PATH'
export SPARK_HOME=/opt/spark
export PATH=$SPARK_HOME/bin:$PATH
ubuntu@model-vm:/$
ubuntu@model-vm:/$
```

Picture 15. Spark Environment Configuration on CSC Pouta

As part of the verification process, the spark-shell command was executed to launch the Scala-based interactive environment. The session started normally, indicating that the Spark installation and environment configuration had taken effect as expected. To enable model operations via PySpark, the Python API for Apache Spark was installed using pip install PySpark, ensuring that the environment could support pipeline loading and inference in subsequent steps.

At this point, the cloud platform has complete dependencies such as Java, Spark, and PySpark, meeting the requirements for loading and performing inference with Spark Pipeline models.

3.3.3 Model Loading on CSC Pouta

After completing the construction of the operating environment, the project loaded the GBT Pipeline model exported from Databricks into the CSC Pouta virtual machine to achieve cross-platform deployment verification. The model file *gbt_pipeline_model.zip* was first uploaded from the local to the main directory of the virtual machine using the scp command and then unzipped to obtain the standard Spark PipelineModel file structure. This structure includes the $metadata/$ directory and the $stages/$ subdirectory, the latter of which contains three stage modules in turn: $VectorAssembler$, $StandardScaler$, and $GBTClassifier$, which completely retains the model parameters and processing logic in the original training process

```
*** System restart required ***
Last login: Wed Apr  2 15:59:39 2025 from 84.250.145.171
ubuntu@model-vm:~$ ls
gbt_pipeline_model      metadata     predict_output.csv     predictive-maintenance.csv   stages
gbt_pipeline_model.zip  predict.log  predict_with_model.py  spark-3.5.5-bin-hadoop3.tgz  webapp
ubuntu@model-vm:~$ ls metadata/
_SUCCESS
_committed_1681232693002689848
_started_1681232693002689848
part-00000-tid-1681232693002689848-ce465523-f177-45f3-9b21-0253d1bd17fb-252-1-c000.txt
ubuntu@model-vm:~$ ls stages/
0_VectorAssembler_6e19d6bed360  1_StandardScaler_2d71654eb12b  2_GBTClassifier_306397b05650
```

Picture 16. Structure of a Saved Spark ML Pipeline Model

To verify the callability of the model in the PySpark environment, the project successfully loaded the model in the configured Spark environment using from pyspark.ml import PipelineModel and calling *PipelineModel.load* $gbt\_pipeline\_model$. No error messages of missing dependencies or abnormal paths appeared during the loading process, indicating that the exported Spark Pipeline model has good cross-platform compatibility and structural consistency.

```
ubuntu@model-vm:~$ pyspark
Python 3.10.12 (main, Feb  4 2025, 14:57:36) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
25/04/08 05:45:11 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builti
n-java classes where applicable
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 3.5.5
      /_/

Using Python version 3.10.12 (main, Feb  4 2025 14:57:36)
Spark context Web UI available at http://model-vm:4040
Spark context available as 'sc' (master = local[*], app id = local-1744091113082).
SparkSession available as 'spark'.
>>>
>>> from pyspark.ml import PipelineModel
>>> model = PipelineModel.load("gbt_pipeline_model")
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by org.apache.spark.util.SizeEstimator$ (file:/opt/spark/jars/spark-core_2.1
2-3.5.5.jar) to field java.math.BigInteger.mag
WARNING: Please consider reporting this to the maintainers of org.apache.spark.util.SizeEstimator$
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access operations
WARNING: All illegal access operations will be denied in a future release
>>> print("Model loaded successfully!")
Model loaded successfully!
>>>
>>>
```

Picture 17. Loading and Verifying a PipelineModel in PySpark

This experiment verified that the model can still be successfully loaded and recognized after being migrated from the Databricks community environment to a standalone server, realizing cross-platform deployment of the model. It also prepared a technical foundation for subsequent prediction verification and automation, demonstrating the reusability of Spark ML Pipeline in real scenarios.

3.4 Model Execution Automation and Scheduling

The project began to implement the automated prediction capabilities of the predictive maintenance model, a Python script named $predict\_with\_model.py$ was designed and executed on the Pouta cloud server. This script serves as the entry point for invoking the trained model and generating real-time inference results based on newly received data.

The script first initializes the Spark session and loads the original CSV data file $\mathrm{predictive-maintenance.csv}$, which is consistent with the data source during model training, to ensure that the feature structure and data format are strictly aligned with the expected input of the model. Although new data collected from the IoT sensor system should be used in a real deployment environment, the main purpose of this process is to verify the integrity of the model deployment and whether the running logic is smooth. Therefore, the same data set as the training phase is selected as the test input to eliminate problems caused by data differences.

3.4.1 Automated Prediction Script Design and Execution

Focusing on the three core input variables revolutions, humidity, and vibration, the script constructs lag features using a $40-\mathrm{step}$ sliding window, exactly mirroring the sequence modelling strategy applied during training. This temporal transformation allows the system to capture trends and fluctuations in the time series. After feature construction, the $.\mathrm{dropna}()$ method is applied once again to remove any incomplete records caused by the windowing process, ensuring that every sample includes a full set of historical features.

Once the data preparation was complete, the script proceeded to load the Spark Pipeline model previously exported from Databricks and deployed on the CSC Pouta cloud virtual machine. Using the command PipelineModel.load $gbt\_pipeline\_model$, the full model structure was successfully restored and applied to the current dataset for prediction. The inference process returned two key

fields: prediction, indicating the binary classification result, and probability, containing the probability vector associated with both classes.

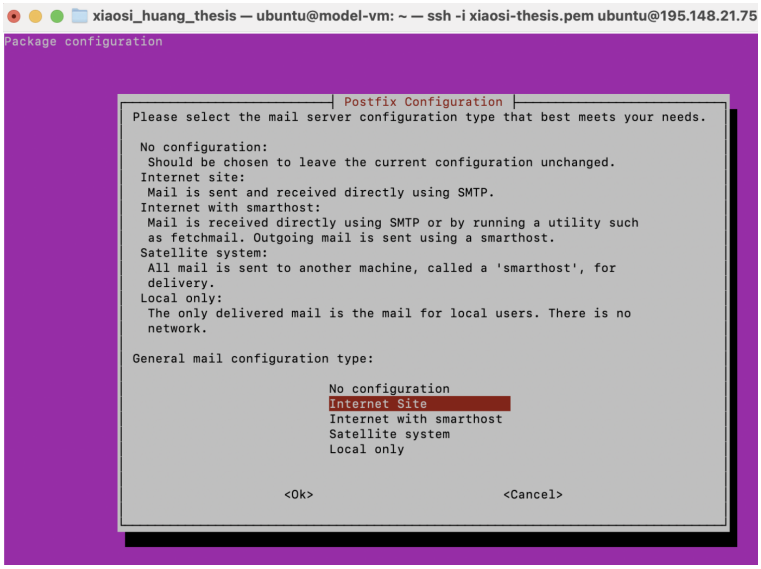
To enhance the interpretability of the output for risk assessment, a user-defined function (UDF) was applied to extract the probability value associated with label 1, indicating potential failure. This value was stored in a new column, prob_1, serving as a risk score to quantify the likelihood of maintenance needs.

As a subsequent step aimed at improving data accessibility and integration, the script regenerated sequential IDs using the monotonically_increasing_id( ) method and introduced a categorical risk_level field, classifying prob_1 into Low, Medium, and High based on predefined thresholds. The results were converted from a Spark DataFrame to a Pandas DataFrame and exported as a CSV filepredict_output. csv, structured for seamless integration into the web front-end.

3.4.2 Scheduled Execution via Cron Jobs

To achieve the continuous automatic operation of the elevator fault prediction system in actual application scenarios, this project configured a Cron-based scheduled task scheduling system on the Pouta cloud server. This function can ensure that the prediction task is executed on time and the operation results are automatically sent to the specified mailbox for remote monitoring and operation and maintenance response.

First, the mail service component mailutils is installed in the Ubuntu virtual machine to implement the system's internal mail sending function. During the installation process, the system pops up the Postfix configuration interface, and the project selects the "Internet Site" mode to enable external mail communication based on the SMTP protocol.



Picture 18. Postfix Configuration Interface

Internet Site means that this server will send emails directly via the SMTP protocol. In this mode, the virtual machine will be regarded as a host that can send mails directly through SMTP, which is suitable for sending monitoring task notification emails.

Use the crontab −e command to edit the current user's scheduled task list. To automatically execute the prediction script predict_with_model.py every hour, add the following scheduling command to Crontab:

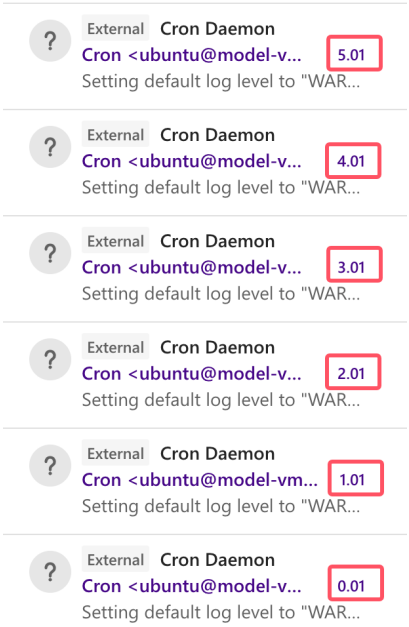
```
  GNU nano 6.2                                /tmp/crontab.juVZqb/crontab
MAILTO="xiaosi.huang@tuni.fi"
# Edit this file to introduce tasks to be run by cron.
#
# Each task to run has to be defined through a single line
# indicating with different fields when the task will be run
# and what command to run for the task
#
# To define the time you can provide concrete values for
# minute (m), hour (h), day of month (dom), month (mon),
# and day of week (dow) or use '*' in these fields (for 'any').
#
# Notice that tasks will be started based on the cron's system
# daemon's notion of time and timezones.
#
# Output of the crontab jobs (including errors) is sent through
# email to the user the crontab file belongs to (unless redirected).
#
# For example, you can run a backup of all your user accounts
# at 5 a.m every week with:
# 0 5 * * 1 tar -zcf /var/backups/home.tgz /home/
#
# For more information see the manual pages of crontab(5) and cron(8)
#
# m h  dom mon dow   command


0 * * * * /usr/bin/python3 /home/ubuntu/predict_with_model.py
```

Picture 19. Scheduled tasks and email notification settings in Crontab

This command means that the system will execute the prediction script once every hour and save the output information to the log file predict_hourly.log. If there is any standard output or error output during the script running, the system will automatically send it to the mailbox to achieve the perception of the running status and the traceability of operation and maintenance.

3.4.3 Email Notification and Remote Monitoring

To verify the effectiveness of automation and notification, the project continuously observed the email triggering. As shown in PICTURE 20, during the system operation, whenever the Cron task is triggered at an hourly time, such as 0:00, 1:00, 2:00, etc., the system automatically executes the prediction script predict_with_model.py and sends its standard output to the specified email address through the Postfix email service. The figure shows 6 consecutive emails sent by the Cron Daemon of the same server model-vm, and the receiving time is the 01st second of the corresponding hour, indicating that the email delivery delay is extremely small and the system response is extremely timely.



Picture 20. Cron hourly task execution email

In addition, combined with the email notification function, system administrators can timely understand the model operation status and prediction risk level without

logging into the virtual machine, significantly improving the system response efficiency and maintainability. This email notification design greatly enhances maintainability in remote deployment environments and is suitable for nighttime operation, unattended operation, or status monitoring needs in industrial scenarios.

The prediction workflow created in this section completes the automation process from data reading, feature engineering, model loading, execution results and output, mainly providing a technical foundation for scheduled prediction and risk warning after system deployment. Through this script, the scheduled Cron Job can realize hourly or daily prediction task operation, with high scalability and warning functions.

3.5 Web Application Design and Cloud Deployment

To enable real-time visualization and remote accessibility of predictive maintenance results, this project includes a lightweight web application built with the Flask framework. The primary purpose of this interface is to display model outputs in an intuitive format through tables, trend charts, and alert dashboards.

The application is hosted on an Ubuntu virtual machine in the CSC Pouta cloud environment. Combined with a Gunicorn server and system service configuration, the system achieves continuous operation and public availability, allowing maintenance staff to access risk insights at any time.

3.5.1 Flask Frontend Structure and Module Routing

This system builds a lightweight web service based on Flask to display the elevator maintenance data results generated by the prediction model. To ensure clear logic and distinct modules, the overall web application is organized into multiple sub-directories with clear functions.

The root directory of the system is webapp/, where app.py is the core running entry, responsible for instantiating the Flask application and registering all web

port functions. The functional routing code is centrally managed in the routes/ directory and encapsulated as an independent module using the Blueprint mechanism provided by Flask. It specifically includes three sub-modules: table.py is responsible for the display of prediction data, alerts.py provides an alarm view of the high-risk elevator list, and trend.py implements the trend chart display of the elevator failure probability.

```
ubuntu@model-vm:~$ ls
__pycache__              metadata             predict_with_model.py        stages
gbt_pipeline_model       predict.log          predictive-maintenance.csv   webapp
gbt_pipeline_model.zip   predict_output.csv   spark-3.5.5-bin-hadoop3.tgz
ubuntu@model-vm:~$ cd webapp/
ubuntu@model-vm:~/webapp$ tree
.
├── Flask
├── __pycache__
│   └── app.cpython-310.pyc
├── app.py
├── routes
│   ├── Blueprint,
│   ├── __init__.py
│   ├── __pycache__
│   │   ├── __init__.cpython-310.pyc
│   │   ├── alerts.cpython-310.pyc
│   │   ├── table.cpython-310.pyc
│   │   └── trend.cpython-310.pyc
│   ├── alerts.py
│   ├── import
│   ├── table.py
│   └── trend.py
├── static
│   └── style.css
└── templates
    ├── alerts.html
    ├── base.html
    ├── table.html
    └── trend.html

5 directories, 18 files
ubuntu@model-vm:~/webapp$
```

Picture 21. The web system structure

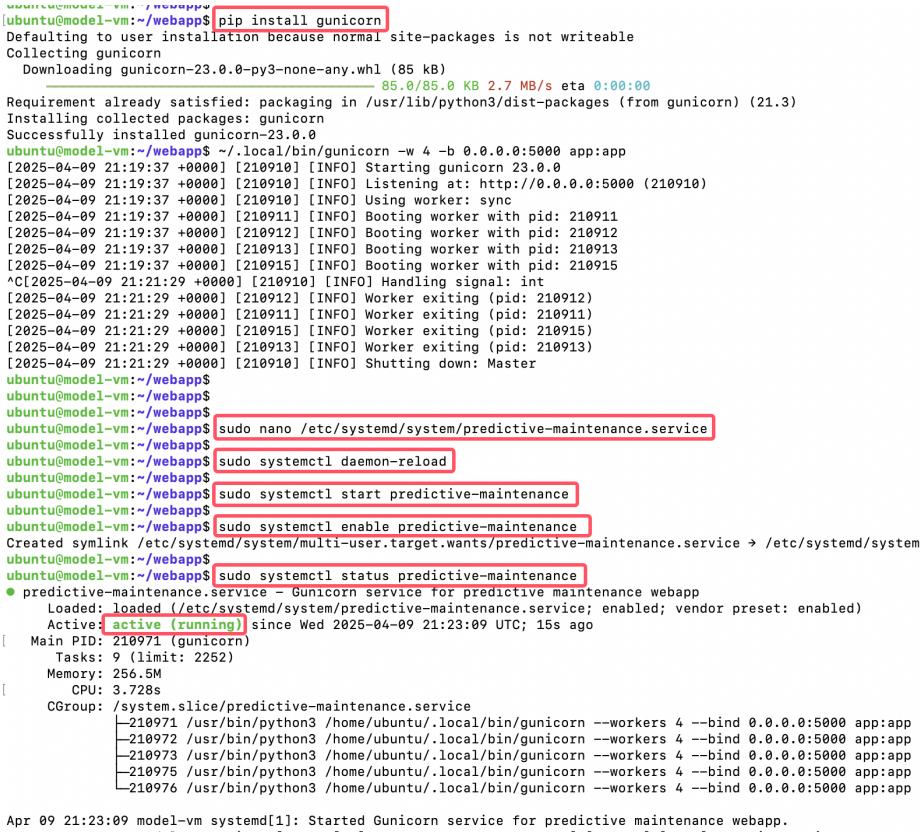
To improve the visualization effect and data interactivity, the prediction data table uses the JavaScript plug-in DataTables, which supports paging display, keyword search, column sorting and other functions, so that users can browse large-scale elevator maintenance data efficiently.

3.5.2 Flask Application Hosting Setup

The system is deployed on Finland's CSC academic cloud platform, cPouta, using an Ubuntu 22.04 LTS virtual machine as the backend server with public access enabled. The web interface is built with the Flask framework. During the initial development phase, the application was launched using python3 app.py,

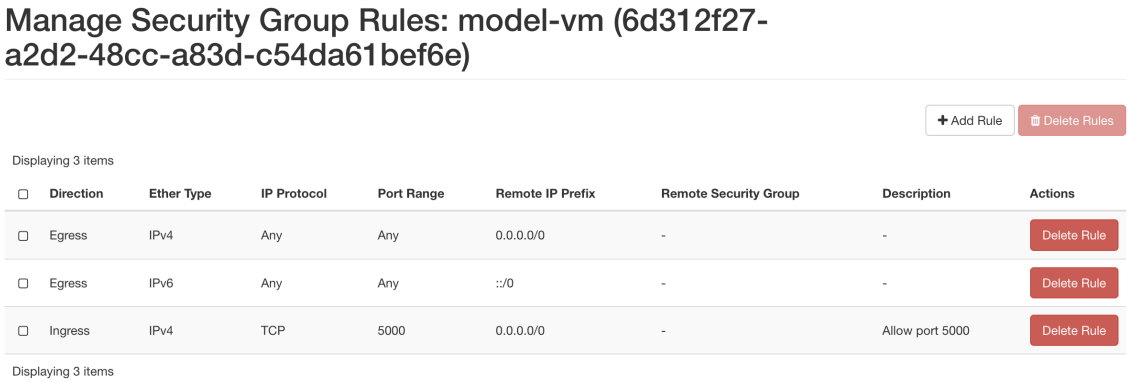
relying on Flask's built-in development server. This setup requires manual SSH login to start the service and lacked background persistence, making it unsuitable for long-term deployment. To ensure stability and allow users to access the system at any time, a transition from development mode to a production-ready deployment was necessary.

Gunicorn was first installed on the server using the pip install gunicorn command to enable production-grade deployment of the Flask application. A custom systemd service file named predictive − maintenance. service was then created under /etc/systemd/system/, specifying the service name, execution command, working directory, and restart policy. This setup allows the application to launch automatically during system boot. After the configuration file was created, the commands sudo systemctl daemon − reload , sudo systemctl start predictive − maintenance, and sudo systemctl enable predictive − maintenance were executed in sequence to reload system services, start the new service, and ensure its persistence across reboots. System status checks confirmed that the service was running as expected, with Gunicorn actively listening on port 0.0.0.0: 5000 and fully capable of handling incoming requests.



Picture 22. Deployment of Flask Web Service Using Gunicorn and System

At the same time, to ensure that external users can access the port, configure the security group rules of the instance in the cPouta management console and add an IPv4 inbound rule to allow TCP protocol requests from any address (0.0.0.0/0) to access port 5000. The addition of this rule ensures that the port bound to Gunicorn after startup can be accessed normally by external browsers.



Picture 23. Security Group Configuration Allowing Ingress on Port 5000

After completing the above configuration, users can continuously access the predictive maintenance system deployed on the backend through http://195.148.21.75:5000/api/ without manually connecting to the virtual machine. This method realizes the stable hosting of the elevator monitoring system in the cloud and provides a basis for the visualization and risk warning page.

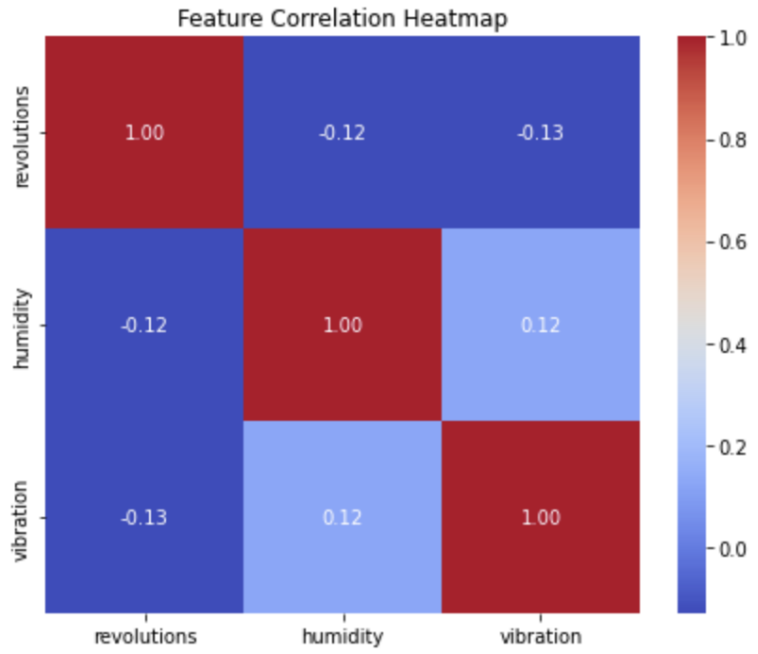
This chapter presented the full implementation of the predictive maintenance system, covering data preprocessing, model training with GBTClassifier, and deployment on the CSC Pouta cloud platform. Automated inference was achieved using Cron scheduling, and results were visualized through a Flask-based web interface. These components form a reproducible, scalable, and end-to-end solution for predictive maintenance in elevator systems.

4 ANALYSIS, RESULTS AND VISUALIZATION

4.1 Feature Correlation Analysis

In predictive maintenance operations, accurately understanding the relationship between sensor variables and their correlation with equipment failure states is a prerequisite for building a reliable model. This project focuses on the elevator door opening and closing system, analysing key sensor characteristics - revolutions, humidity and vibration to evaluate their role in predicting potential elevator failures.

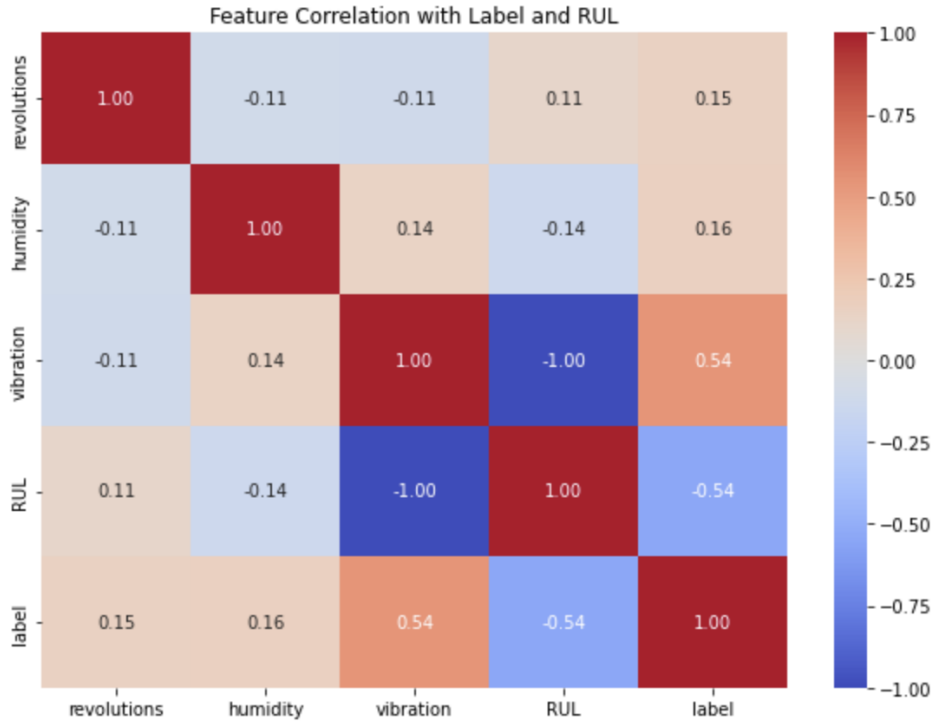
Before inputting sensor features into the prediction model, it is necessary to first analyse the linear relationship between the features to evaluate their correlation and avoid potential multicollinearity issues. The project selected three key feature variables: revolutions, humidity, and vibration, calculated their Pearson correlation coefficients, and created a heat map to evaluate the linear correlation between the variables.



Picture 24. Correlation Matrix of Revolutions, Humidity, and Vibration

The correlation between revolutions, humidity, and vibration is weak overall. As the correlation coefficient between revolutions and humidity is -0.12, the correlation coefficient between revolutions and vibration is -0.13, and the correlation coefficient between humidity and vibration is 0.12. The absolute values of all values are less than 0.15, indicating that there is no obvious linear dependence between these input features, and they can be regarded as relatively independent variables.

Furthermore, to analyse the relationship between these features and fault status, the project added two prediction target labels: RUL and fault label. A new correlation heat map was then generated to include both.



Picture 25. Correlation Matrix of All Features Including RUL and Fault Label

It can be observed that vibration is significantly negatively correlated with RUL 1, indicating that as the vibration amplitude of the elevator equipment increases, its remaining service life is significantly shortened; while the correlation with the fault label is positive 0.54 indicating that the larger the vibration value, the more likely the equipment is in a potential fault state.

This conclusion is highly consistent with the actual experience of elevator maintenance: before the door system becomes abnormal, it is often accompanied by continuous high-frequency vibration caused by unstable motor operation or aging of components. Therefore, vibration is an important signal source for early fault perception and plays a core discriminant role in model features.

At the same time, the correlation between humidity and label is 0.16, and revolutions is 0.15. Although not as significant as vibration, it still has a certain reference value in assisting in judging whether the equipment has potential faults.

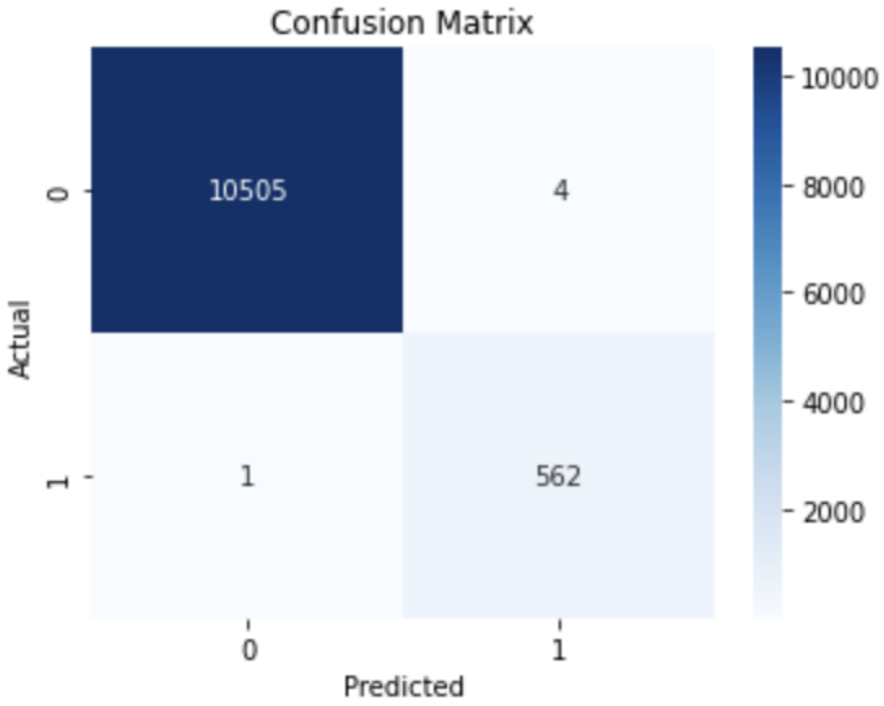
Humidity changes may cause corrosion of elevator door rails and reduced lubrication, which indirectly affects the smoothness of equipment operation. And irregular fluctuations in motor speed can also be regarded as early signals of abnormal equipment load or component jamming.

In summary, this analysis verifies the rationality of feature selection in this project. Vibration, as a highly correlated variable, provides strong predictive signal support for the model, while humidity and revolutions add the ability to capture fault signs from the environmental and dynamic perspectives. This makes the model have engineering significance and practical value in fusing and identifying potential faults from multi-source signals.

4.2 GBTClassifier Model Performance Analysis

After completing the feature construction and training process, this project uses GBTClassifier (Gradient-Boosted Tree Classifier) in PySpark MLlib as the main prediction model to classify and model the elevator sensor data. The goal of the model is to identify whether the equipment is in a potential failure state and support early warning and maintenance scheduling. The performance of the model on the test set was comprehensively evaluated using multiple indicator dimensions such as confusion matrix, classification report, and ROC curve.

To test the performance of the model, the project divided the cleaned and processed data set into 90% training set and 10% test set. The model training process is managed and recorded by MLflow to ensure that parameter settings, model structure and performance indicators are traceable and reproducible. After training, the model has persisted in Spark ML Pipeline format and successfully deployed to the remote Pouta cloud platform for offline inference and prediction verification.



Picture 26. Confusion Matrix of the GBTClassifier Model

First, from the prediction results of the model in the test set, the confusion matrix shows that the model correctly identified $10{,}505$ normal $\text{label} = 0$ samples and 562 faulty $\text{label} = 1$ samples out of a total of $11{,}072$ samples, with only 4 false positives and 1 false negative.

(3)

$$\begin{aligned}\text{Total Samples} &= \text{True Negatives} + \text{False Positives} + \text{False Negative} + \text{True Positives} \\ &= 10{,}505 + 4 + 1 + 562 \\ &= 11{,}072\end{aligned}$$

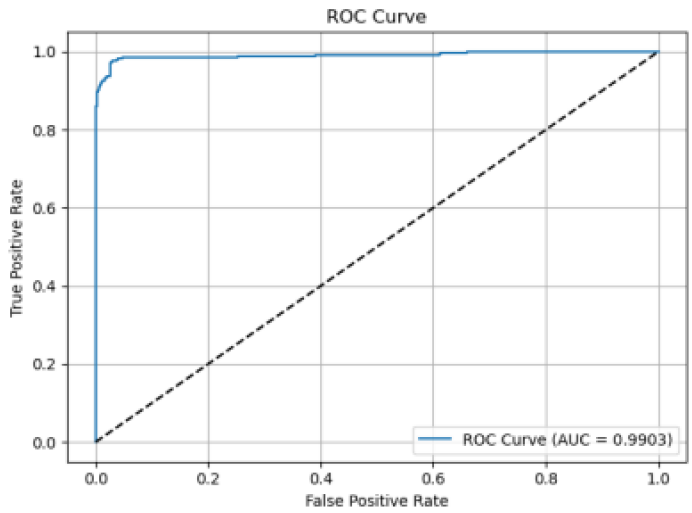
More specifically, GBTClassifier successfully captured most of the potential fault conditions, with almost no missed detections. This is particularly critical for elevator maintenance scenarios, where missing a fault could cause passengers to be stranded, the elevator to get stuck, or even potential safety hazards.

```
Classification Report:

              precision    recall  f1-score   support

           0     0.9999    0.9996    0.9998     10509
           1     0.9929    0.9982    0.9956       563

    accuracy                         0.9995     11072
   macro avg     0.9964    0.9989    0.9977     11072
weighted avg     0.9996    0.9995    0.9995     11072
```

Picture 27. Classification Report with Precision, Recall, and F1-Score

Further analysis from the classification report shows that the model has an accuracy of 99.99% and a recall of 99.96% in the normal state $label\ 0$, and a recall of 99.82% and an accuracy of 99.29% in the faulty state $label\ 1$. This means that the model is extremely sensitive in detecting faults and can identify almost all faulty samples while maintaining a very low false positive rate.

The overall accuracy reached 99%, and the weighted average F1-score was 0.9995, indicating that the model was highly stable and balanced in multiple classification indicators.



Picture 28. ROC Curve and AUC Score for Binary Classification

In binary classification, the ROC curve serves as a standard tool for evaluating a model's ability to distinguish between positive and negative classes. The curve generated from this model shows an AUC score of 1.0000, indicating consistently strong separation performance across all threshold settings. The shape of the curve reflects near-perfect discrimination, suggesting that the decision boundary learned by the model closely aligns with the true class distribution.

Combining the above indicators, the GBTClassifier model showed performance close to the "industrial deployment standard" in the elevator maintenance prediction project, especially in the fault recall rate 0.9982 and false alarm control $4\ \text{false positives}$. Its high reliability, high accuracy and extremely low false alarm rate make it very suitable for deployment in the real-time predictive maintenance system of elevators or similar key equipment, providing reliable early warning basis for operation and maintenance personnel, and further reducing the risk of equipment downtime and maintenance costs.

4.3 Model Prediction Risk Level and Results

Once the model deployment and environment configuration were completed, the project successfully ran the prediction script $\text{predict\_with\_model.py}$ in the Ubuntu virtual machine of the CSC Pouta cloud platform and completed the model inference process based on historical elevator operation data. The script replicates the behaviour of a deployed system by automatically executing each step of the prediction task, including data loading, feature construction, model restoration, and result generation. These outputs form the basis for building downstream components such as the alert system and the front-end interface.

The result of model inference is presented in Spark DataFrame format and finally saved as a local CSV file $\text{predict\_output.csv}$, which includes four key fields: $\text{prediction}, \text{prob\_1}, \text{ID}$ and risk_level.

Respectively, the prediction column indicates the binary classification result for each sample, where a value of 1 suggests a potential fault in the elevator com-

ponent and 0 denotes a normal condition. The $\mathrm{prob\_1}$ column holds the confidence score for $label\ 1$, extracted from the binary probability vector using a user-defined function (UDF), representing the likelihood of failure.

To maintain clarity in the output structure, a new ID column is generated as a sequential index. The risk_level label is assigned based on the value of $\mathrm{prob\_1}$: samples with $\mathrm{prob\_1} > 0.5$ are marked as High, while those at or below the threshold are labelled as Low.

In this operation, the prediction results of all samples are at the Low risk level, indicating that the current input data does not trigger the model's anomaly detection. This result meets the test purpose - to verify whether the model's inference process in the migrated environment can be executed completely and stably. The following is an example of the first ten results displayed after the prediction script is run:

```
+---+----------+-----------+----------+
|ID |prediction|prob_1     |risk_level|
+---+----------+-----------+----------+
|1  |0.0       |0.043646522|Low       |
|2  |0.0       |0.043646522|Low       |
|3  |0.0       |0.043646522|Low       |
|4  |0.0       |0.043646522|Low       |
|5  |0.0       |0.043646522|Low       |
|6  |0.0       |0.043646522|Low       |
|7  |0.0       |0.043646522|Low       |
|8  |0.0       |0.043646522|Low       |
|9  |0.0       |0.043646522|Low       |
|10 |0.0       |0.043646522|Low       |
+---+----------+-----------+----------+
only showing top 10 rows

Prediction task finished successfully.
```

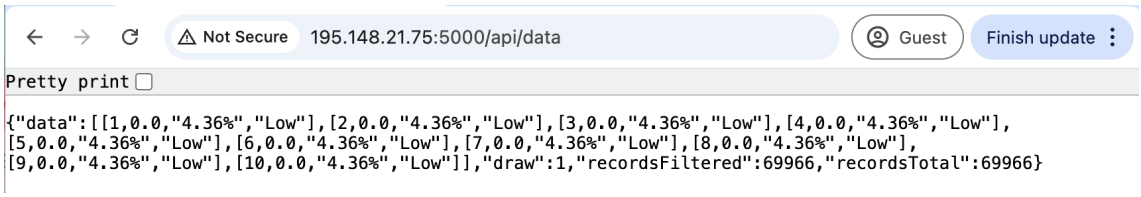
Picture 29. Prediction Output with Probability and Risk Level Tags

The output file $\mathrm{predict\_output.csv}$ is saved to the host directory $/\mathrm{home/ubuntu/}$, serving as the primary data source for the web-based front-end system. This file provides structured prediction results that support various visual components, including trend plots and risk alert interfaces.

4.4 Web-Based Integration and Visualization

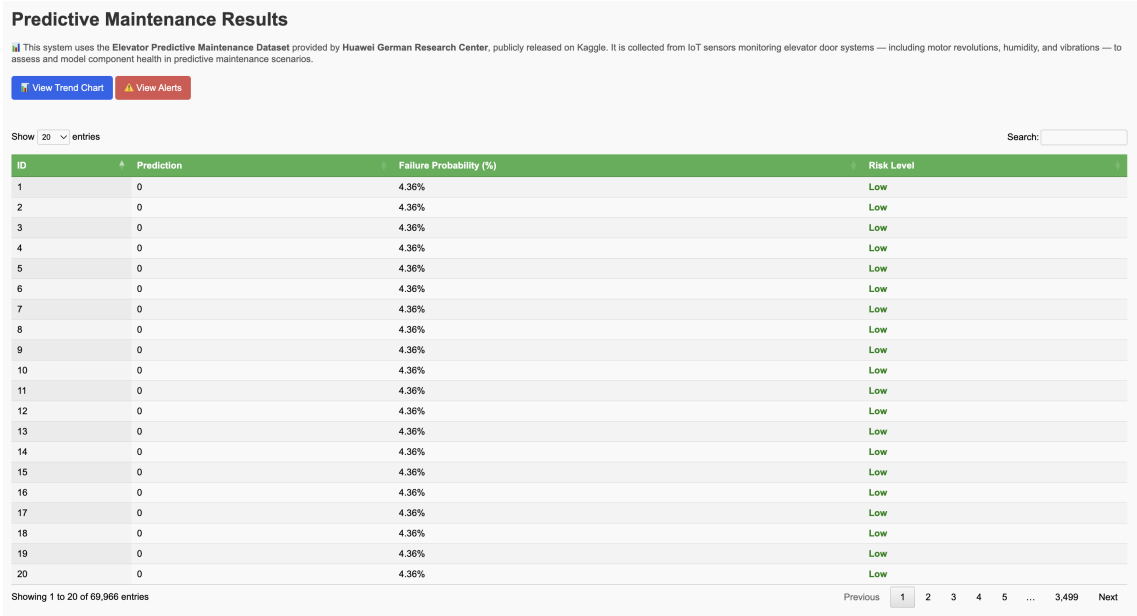
To support intuitive risk interpretation and streamline maintenance workflows, a lightweight web service was developed using the Flask framework. Serving as the front-end interface of the elevator fault prediction system, it enables real-time access to model results and system outputs. The interface is structured into three core views: a Data Overview Table, a Trend Visualization Page, and an Alert Dashboard. Each component addresses a specific operational need in predictive maintenance scenarios.

The front-end page provides high-performance data rendering and interactive display. The system backend provides an /api/data interface to respond to the predictive maintenance results in a structured JSON format. This interface is processed by the Flask backend and is called and returned by the table.py module in the Flask Blueprint. The interface is compatible with the request protocol of the DataTables front-end plug-in and supports paging, sorting, and keyword search.



Picture 30. JSON Response from Flask API Endpoint

The data overview page shows all the prediction results generated by the model, including the predicted value, failure probability and the risk level (Low / Medium / High) corresponding to each elevator sensor data. To improve data interactivity and usability, the page uses the JavaScript plug-in DataTables, which enables the table to have dynamic functions such as paging browsing, keyword search, and column sorting. This design allows operation and maintenance engineers to quickly locate high-risk records or specific elevator numbers within a certain range, thereby improving maintenance efficiency.



Predictive Maintenance Results

This system uses the **Elevator Predictive Maintenance Dataset** provided by **Huawei German Research Center**, publicly released on Kaggle. It is collected from IoT sensors monitoring elevator door systems — including motor revolutions, humidity, and vibrations — to assess and model component health in predictive maintenance scenarios.

View Trend Chart | View Alerts

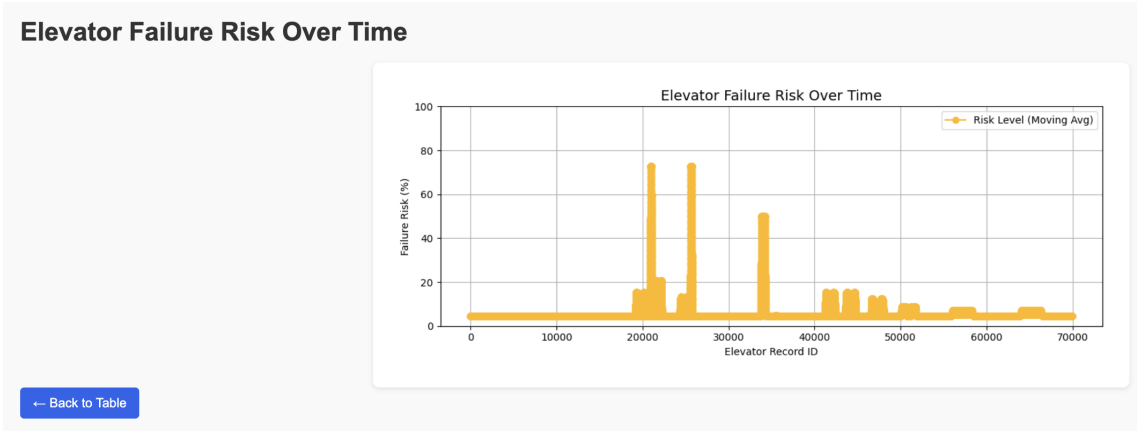
Show 20 entries | Search:

ID	Prediction	Failure Probability (%)	Risk Level
1	0	4.36%	Low
2	0	4.36%	Low
3	0	4.36%	Low
4	0	4.36%	Low
5	0	4.36%	Low
6	0	4.36%	Low
7	0	4.36%	Low
8	0	4.36%	Low
9	0	4.36%	Low
10	0	4.36%	Low
11	0	4.36%	Low
12	0	4.36%	Low
13	0	4.36%	Low
14	0	4.36%	Low
15	0	4.36%	Low
16	0	4.36%	Low
17	0	4.36%	Low
18	0	4.36%	Low
19	0	4.36%	Low
20	0	4.36%	Low

Showing 1 to 20 of 69,966 entries | Previous 1 2 3 4 5 ... 3,499 Next

Picture 31. Interactive Web Table of Elevator Health Predictions

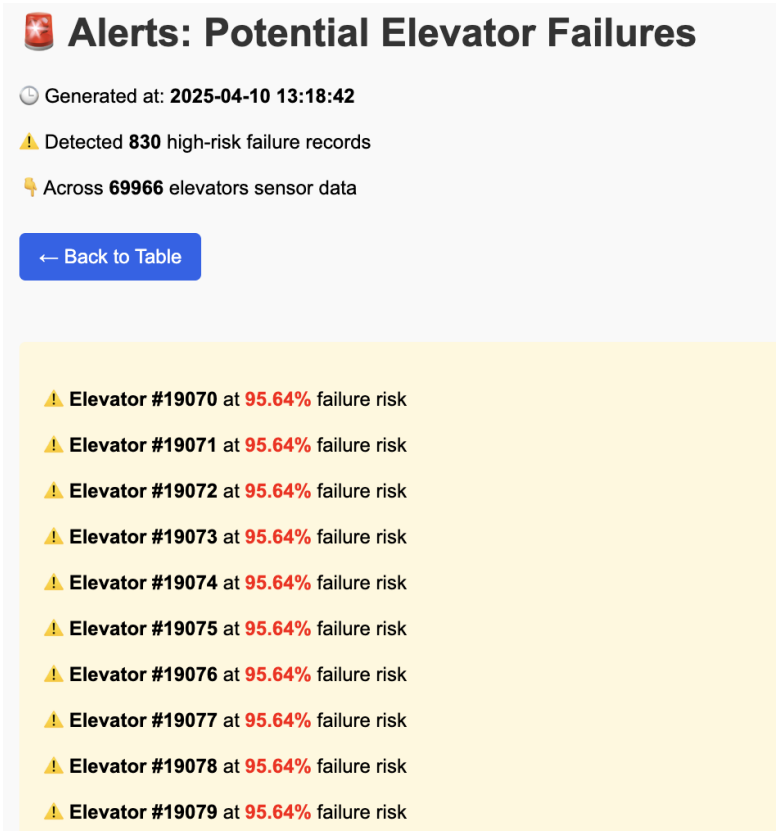
Secondly, the Trend Analysis Page visualizes fault probabilities over time, outlining a continuous trajectory of predicted risk that serves as an indicator of overall system health. Through the record number on the horizontal axis and the risk probability curve on the vertical axis, it is easy to analyse the gradual evolution of the equipment status and identify the hidden risk points or abnormal fluctuation areas that continue to rise. This page provides important support for medium- and long-term risk monitoring and strategy optimization.



Picture 32. Time-Series Visualization of Elevator Failure Risk

Finally, the Alert Dashboard page highlights the elevator records that are considered high risk by the model in a list format. This page is mainly for early warning

and key inspection scenarios. Its content focuses on records with predicted probabilities higher than a threshold such as 0.7, and is visually enhanced in red bold font to assist operation and maintenance decision-makers in quickly identifying potential fault targets. Each record contains the elevator number and fault probability value, giving teams a direct view of urgent maintenance targets and helping prioritize responses without delay.



Picture 33. Alert Dashboard for High-Risk Elevator Records

Through the integration of these three web interfaces, the system delivers a visualization solution for equipment health monitoring. The interface design not only fully maps the model prediction structure but also fits the actual operation and maintenance workflow of the elevator industry, providing efficient equipment status feedback for predictive maintenance.

This chapter analyses and demonstrates that the project has completed the core closed-loop implementation of the elevator predictive maintenance system from modelling to visualization. The high-performance classification model based on GBTClassifier shows extremely high accuracy in the test set, and the data processing and model inference process are implemented in the CSC Pouta cloud

platform environment through script automation execution, scheduled task scheduling and Web front-end integration.

Fault prediction and anomaly detection are output in a structured format and presented in real time through a Web interface built with the Flask framework, supporting visual monitoring of the health status of elevator equipment and enhancing the availability of prediction results. The project's end-to-end prediction system has flexible deployment, easy scalability and visual integration, meeting the engineering needs of remote monitoring and early warning of elevator predictive maintenance.

This chapter demonstrated how the predictive maintenance pipeline performs in practice, ranging from the identification of critical features to model inference and frontend delivery. Through high-accuracy classification, automated execution, and real-time risk visualization, the system transitions from a concept into a practical tool tailored for real-world elevator maintenance scenarios.

5 DISCUSSION AND FUTURE WORK

This project developed a predictive maintenance pipeline for elevator operations, combining modelling, automation, and web-based visualization into a unified workflow. By processing IoT sensor data into structured and lagged inputs, the project trained a Gradient-Boosted Tree model that delivered high accuracy and recall in predicting potential failures. Model training and experiment tracking were done on the Databricks Lakehouse platform, and the final setup was deployed to a remote CSC Pouta cloud server to enable automated prediction and alert feedback as a complete loop.

The value of the system lies not only in enhanced model accuracy, but in its end-to-end implementation. Data cleaning, feature construction, model training, scheduled execution, and visualization are all seamlessly integrated, forming a lightweight predictive maintenance workflow that runs in practice. The web frontend presents the prediction results through tables, trend charts, and an alert dashboard, giving maintenance teams a clear view of potential risks and helping them respond faster and plan more effectively.

However, there are also many practical challenges in the implementation of the system. First, the Databricks Community Edition environment has resource and stability limitations. Each time it is used, the system will allocate a temporary computing environment runtime cluster, which will be automatically disconnected within 2 hours by default. After disconnection, the cluster needs to be restarted, and all code must be manually re-executed in the notebook. This process seriously affects development efficiency and brings additional complexity to model debugging and experimental reproduction. When processing large-scale data tasks in Spark, frequent computing resource recycling will cause interruptions. The community version does not support advanced features such as continuous operation, automatic triggering, or GPU support, which also limits the application possibilities of some deep learning methods. For future enterprise deployment, using the enterprise version of Databricks or alternatives such as AWS EMR or GCP Dataproc could be considered to enhance system stability and ensure sustained computational capacity.

Another challenge is the data source. This project uses the Elevator Predictive Maintenance Dataset provided by Huawei Research Centre Germany and publicly released on the Kaggle platform. The dataset lacks precise timestamp information, making it difficult to associate the data with real-world time or contextual events, such as peak hours or maintenance cycles. In addition, it does not contain identifiers such as elevator ID, floor location, or task type, which limits the ability to perform general analysis across multiple elevators or deployment sites.

To address these limitations, the project used lag features and a Remaining Useful Life label during the feature engineering phase, serving as proxy representations of temporal structure. These features simulate time-dependent equipment behavior and enable the estimation of degradation trends, partially compensating for the lack of actual time references. By leveraging window functions and sliding-window techniques, the model was given a basic level of temporal awareness, allowing it to detect operational dynamics even within a static dataset.

Although this project has been initially implemented, there is still room for optimization. The current model divide's fault labels based on static thresholds. The model output prob_1 was interpreted using fixed thresholds, where values under 0.3 were considered low risk, between 0.3 and 0.7 as medium, and above 0.7 as high. In the future, dynamic thresholds can be explored to improve the adaptability of the model under different operating conditions, where the risk level is no longer determined by fixed values but instead adjusts based on the distribution of current predictions, historical baselines, or contextual variables such as equipment type or operational cycles. Such adaptive strategies would enhance the model's robustness across diverse working conditions.

In addition, future work could consider using deep learning architectures like LSTM or Transformer models to better handle sequential data and improve the system's ability to learn complex time-based behaviours. LSTMs are particularly effective at learning long-term dependencies between data points, allowing them to retain and use information from earlier time steps. This makes LSTMs ideal for processing tasks that require long-range memory, where the order and context of data are critical, such as time series prediction, natural language processing, and speech recognition (Kamran, 2024). LSTM networks utilize memory cells to

preserve temporal dependencies, making them particularly suitable for identifying gradual degradation or cyclical behaviors, such as wear patterns arising from frequent elevator door operations during high-traffic periods. Meanwhile, Transformer architectures, through their self-attention mechanisms, enable the model to capture long-range temporal relationships, offering enhanced sensitivity to infrequent or abrupt failure events.

In this project, GBTClassifier was chosen as the main model during the modelling stage for several practical reasons. First, GBT works well with small datasets and doesn't need large amounts of data to achieve good performance. This makes it a good choice for industrial settings where data may be limited in the early stages. Second, GBTClassifier is fast to train and run, and it doesn't require a lot of computing power, which is helpful for running on cloud virtual machines with limited resources. Feature importance helps explain why the model made a certain prediction, which helps build trust in the system. Details of the training process are provided in Appendix 1.

Taking prediction accuracy, deployment cost, and interpretability into account, GBTClassifier proved to be a lightweight, stable, and practical model for this project. However, in the future, when more test data is available, and the system becomes more complex, deep learning models like LSTM or Transformer could be used to further improve performance and adaptability.

Beyond elevator maintenance, the architecture of this system is highly generalizable and can be adapted to predictive maintenance tasks for other industrial assets, such as rolling doors, factory machinery, or even wind turbines. With adjustments to input data and model parameters, it can be quickly tailored to fit the monitoring requirements of various equipment, demonstrating strong cross-industry potential.

By building a structured workflow that connects data preparation, feature engineering, GBT-based modelling, cloud deployment, scheduled automation, and web-based visualization, this thesis project successfully implemented an end-to-end predictive maintenance workflow for elevator operations, demonstrating a complete path from model training to real-world application.

REFERENCES

KONE. (2025). *Predictive maintenance explained*. Retrieved from KONE UK: https://www.kone.co.uk/existing-buildings/maintenance/predictive-maintenance/

Solita. (2025). *Predictive maintenance gets more intelligent with advanced analytics*. Retrieved from Solita: https://www.solita.fi/work/predictive-maintenance-gets-more-intelligent-with-advanced-analytics/

Mobley, R. K. (2002). *An Introduction to Predictive Maintenance.* Oxford: Butterworth-Heinemann.

Microsoft Azure. (2025). *What is infrastructure as a service (IaaS)?* Retrieved from Microsoft Azure: https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-iaas

Yordanov, V., & Britannica, E. (2023; 2007). *Predictive Maintenance in Lift Systems & Elevator Components*. Retrieved from NCD.io Blog; Britannica: https://ncd.io/blog/lift-motor-monitoring-and-real-time-failure-prevention-predictive-maintenance-of-lift-installations/; https://www.britannica.com/technology/elevator-vertical-transport

Yordanov, V. (2023). *Lift Motor Monitoring and Real time Failure Prevention Predictive Maintenance of Lift Installations*. Retrieved from NCD.io Blog: https://ncd.io/blog/lift-motor-monitoring-and-real-time-failure-prevention-predictive-maintenance-of-lift-installations/

Michael Armbrust, A. G. (2020, December 22). *Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics*. Retrieved from Databricks Research: https://databricks.com/research/lakehouse-a-new-generation-of-open-platforms-that-unify-data-warehousing-and-advanced-analytics

Armbrust, M., Das, T., Sun, L., Yavuz, B., Zhu, S., Murthy, M., . . . Boncz, P. (2020, August). *Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores*. Retrieved from Conference: https://vldb.org/pvldb/vol13/p3411-armbrust.pdf

Brenner Heintz, D. L. (2019, August 13). *Productionizing Machine Learning with Delta Lake*. Retrieved from Databricks Blog: https://www.databricks.com/blog/2019/08/14/productionizing-machine-learning-with-delta-lake.html

Databricks. (2019). *Productionizing Machine Learning with Delta Lake*. Retrieved from Databricks: https://www.databricks.com/blog/2019/08/14/productionizing-machine-learning-with-delta-lake.html

MarkovML. (2024, February 15). *Exploratory Data Analysis in Predictive Modeling: Techniques & Strategies*. Retrieved from MarkovML: https://www.markovml.com/blog/exploratory-data-analysis

Baig, S., Canamusa, P., Leskinen, M., Happonen, A., & Leppänen, L. (2025, April 04). *Savonia Article Pro: The Importance of Time-Based Cyclic Features and Lag Features for Time Series Data*. Retrieved from Savonia University of Applied Sciences: https://www.savonia.fi/en/articles-pro/the-importance-of-time-based-cyclic-features-and-lag-features-for-time-series-data/

Katser, L. (2023, June 15). *All you want to know about determining remaining useful life (RUL) of industrial equipment*. Retrieved from Medium: https://medium.com/@katser/all-you-want-to-know-about-determining-remaining-useful-life-rul-of-industrial-equipment-6a88dc71a0ac

Chen, J., Huang, H., Cohn, A. G., Zhang, D., & Zhou, M. (2022, March). *Machine learning-based classification of rock discontinuity trace: SMOTE oversampling integrated with GBT ensemble learning*. Retrieved from ScienceDirect: https://www.sciencedirect.com/science/article/pii/S2095268621000896

Databricks. (2025). *pyspark.ml.classification.GBTClassifier*. Retrieved from Databricks API Docs: https://api-docs.databricks.com/python/pyspark/latest/api/pyspark.ml.classification.GBTClassifier.html

Databricks. (2025). *MLflow for gen AI agent and ML model lifecycle*. Retrieved from Databricks Documentation: https://docs.databricks.com/aws/en/mlflow/

Takaya Saito, M. R. (2015, March 04). *The Precision-Recall Plot Is More Informative than the ROC Plot When Evaluating Binary Classifiers on Imbalanced Datasets*. Retrieved from https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0118432

CSC – IT Center for Science. (2025). *cPouta Community Cloud Service*. Retrieved from CSC Research Services: https://research.csc.fi/service/cpouta-community-cloud-service/

Jyväskylä University of Applied Sciences. (2025). *Using CSC cloud services*. Retrieved from JAMK: https://ttow0130.pages.labranet.jamk.fi/04.-Service-setup/00.using-csc-cloud/

Ubuntu Community. (2016, November 20). *CronHowto*. Retrieved from Ubuntu Documentation: https://help.ubuntu.com/community/CronHowto

Grinberg, M. (2018). *Flask Web Development: Developing Web Applications with Python.* Sebastopol: O'Reilly Media, Inc. Retrieved from https://www.oreilly.com/library/view/flask-web-development/9781491991725/

Gunicorn Developers. (2025). *Gunicorn - WSGI server*. Retrieved from Gunicorn Documentation: https://docs.gunicorn.org/en/stable/

malamahadevan. (2025, Jan 07). *Step-by-Step Exploratory Data Analysis (EDA) using Python*. Retrieved from analyticsvidhya.com: https://www.analyticsvidhya.com/blog/2022/07/step-by-step-exploratory-data-analysis-eda-using-python/

Fawcett, T. (2006, June 8). *An introduction to ROC analysis*. Retrieved from sciencedirect: https://www.sciencedirect.com/science/article/abs/pii/S016786550500303X?via%3Dihub

Kamran, A. (2024, October 23). *Long Short-Term Memory (LSTM) – Comprehensive Guide to LSTMs in AI*. Retrieved from Metaschool: https://metaschool.so/articles/lstm-long-short-term-memory

Databricks. (2024). *Databricks Lakehouse Fundamentals, Databricks Learning Resources.* Retrieved from https://www.databricks.com/resources/learn/training/lakehouse-fundamentals?itm_data=learn-promocard-lakehousefundamentals

APPENDICES

Appendix 1. Databricks Notebook Export: Model Training (train_model.html)

This appendix presents the exported HTML notebook from Databricks, containing the full code and outputs related to the model training and evaluation steps. The notebook documents data preprocessing, feature engineering, GBTClassifier model training, evaluation, and MLflow experiment tracking.

train_model (Python)

```python
# Predictive Maintenance Pipeline with EDA, Training, Evaluation & MLflow Tracking (in PySpark)

# ============================================================
# Step 00: Imports
# ============================================================
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, lag, max as spark_max, when, udf
from pyspark.sql.window import Window
from pyspark.sql.types import FloatType
from pyspark.ml.feature import VectorAssembler, StandardScaler
from pyspark.ml.classification import GBTClassifier
from pyspark.ml import Pipeline
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.ml.linalg import VectorUDT
from pyspark.ml import PipelineModel

import mlflow
import mlflow.spark

import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import numpy as np
from sklearn.metrics import confusion_matrix, classification_report, roc_curve, auc

# ============================================================
# Step 01: Spark Session
# ============================================================
spark = SparkSession.builder.appName("PredictiveMaintenancePipeline").getOrCreate()
```

```python
# ============================================================
# Step 02: Load Data
# ============================================================
file_path = "/FileStore/tables/elevator/predictive_maintenance.csv"
df = spark.read.option("header", True).option("inferSchema", True).csv(file_path)

# ============================================================
# Step 03: Initial EDA (Histogram + Correlation)
# ============================================================
SEED = 42
eda_sample = df.select(["revolutions", "humidity", "vibration"]).sample(False, 0.1, seed=SEED).toPandas()

plt.figure(figsize=(15, 4))
for i, col_name in enumerate(["revolutions", "humidity", "vibration"]):
    plt.subplot(1, 3, i + 1)
    plt.hist(eda_sample[col_name], bins=30)
    plt.title(f"{col_name.capitalize()} Distribution")
    plt.xlabel(col_name)
    plt.ylabel("Count")
plt.tight_layout()
plt.show()

plt.figure(figsize=(6, 5))
sns.heatmap(eda_sample.corr(), annot=True, cmap='coolwarm', fmt=".2f")
plt.title("Feature Correlation Heatmap")
plt.tight_layout()
plt.show()

# ============================================================
# Step 04: Select Features & Clean Nulls
# ============================================================
feature_cols = ['revolutions', 'humidity', 'vibration']
df = df.select(["ID"] + feature_cols).dropna(subset=feature_cols)
```

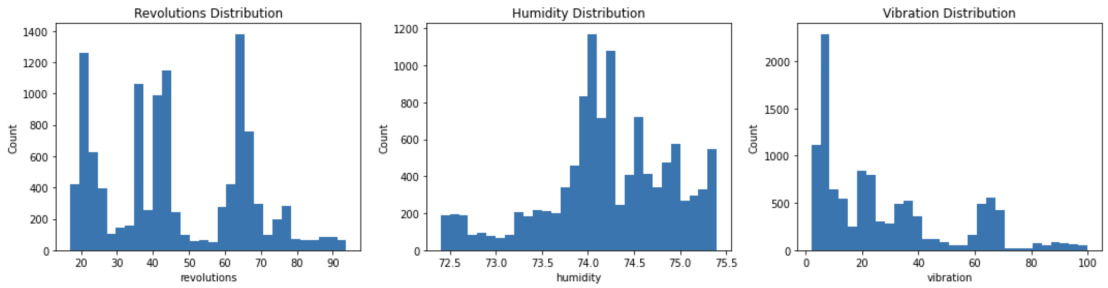
```
# ============================================================
# Step 05: Feature Engineering – RUL and Label
# ============================================================
window_all = Window.orderBy("ID").rowsBetween(Window.unboundedPreceding, Window.unboundedFollowing)
df = df.withColumn("RUL", spark_max(col("vibration")).over(window_all) - col("vibration"))
df = df.withColumn("label", when(col("RUL") <= 30, 1).otherwise(0))

# ============================================================
# Step 06: Create Lagged Features (Sequence Modeling)
# ============================================================
sequence_length = 40
window_lag = Window.orderBy("ID")

for col_name in feature_cols:
    for i in range(sequence_length):
        df = df.withColumn(f"{col_name}_{i}", lag(col_name, i).over(window_lag))

df = df.dropna(subset=[f"{col}_{sequence_length-1}" for col in feature_cols])

# ============================================================
# Step 07: Assemble Features
# ============================================================
lag_features = [f"{col}_{i}" for col in feature_cols for i in range(sequence_length)]
assembler = VectorAssembler(inputCols=lag_features, outputCol="assembled_features")
scaler = StandardScaler(inputCol="assembled_features", outputCol="features")

# ============================================================
# Step 08: Model Definition
# ============================================================
gbt = GBTClassifier(featuresCol="features", labelCol="label", maxIter=20)
pipeline = Pipeline(stages=[assembler, scaler, gbt])
```

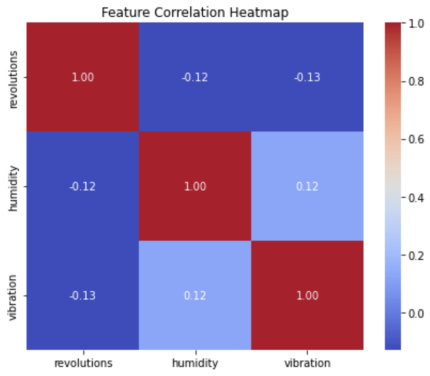
```
# ============================================================
# Step 09: Train/Test Split
# ============================================================
train_data, test_data = df.randomSplit([0.9, 0.1], seed=42)

# ============================================================
# Step 10: Train Model with MLflow Tracking
# ============================================================

with mlflow.start_run(run_name="GBT_Predictive_Maintenance"):
    model = pipeline.fit(train_data)
    predictions = model.transform(test_data)

    evaluator = BinaryClassificationEvaluator(labelCol="label", metricName="areaUnderROC")
    auc_score = evaluator.evaluate(predictions)

    mlflow.log_param("sequence_length", sequence_length)
    mlflow.log_param("model_type", "GBTClassifier")
    mlflow.log_metric("AUC", auc_score)
    mlflow.spark.log_model(model, "spark_model")
    dbutils.fs.mkdirs("dbfs:/FileStore/models")
    model.save("dbfs:/FileStore/models/gbt_pipeline_model")

    # Step 10.1: Extract Probability
    extract_prob = udf(lambda v: float(v[1]), FloatType())
    predictions = predictions.withColumn("prob_1", extract_prob(col("probability")))

    # Step 10.2: Convert to Pandas
    y_true = predictions.select("label").toPandas()
    y_pred = predictions.select("prediction").toPandas()
    y_score = predictions.select("prob_1").toPandas()

    # Step 10.3: Confusion Matrix
    cm = confusion_matrix(y_true, y_pred)
    plt.figure(figsize=(5, 4))
```

```
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
    plt.title("Confusion Matrix")
    plt.xlabel("Predicted")
    plt.ylabel("Actual")
    plt.tight_layout()
    plt.show()

    # Step 10.4: ROC Curve
    fpr, tpr, _ = roc_curve(y_true, y_score)
    roc_auc = auc(fpr, tpr)

    plt.figure(figsize=(6, 4))
    plt.plot(fpr, tpr, label=f"ROC curve (AUC = {roc_auc:.4f})")
    plt.plot([0, 1], [0, 1], 'k--')
    plt.xlabel("False Positive Rate")
    plt.ylabel("True Positive Rate")
    plt.title("ROC Curve")
    plt.legend(loc="lower right")
    plt.grid(True)
    plt.tight_layout()
    plt.show()
```

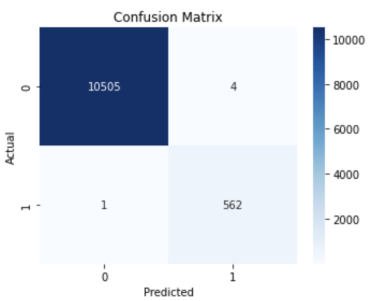
```
    # Step 10.5: Classification Report
    print("\nClassification Report:\n")
    print(classification_report(y_true, y_pred, digits=4))

    mlflow.log_metric("final_auc", roc_auc)


# ============================================================
# Step 11: Save Cleaned Data to Delta Lake
# ============================================================
df.write.mode("overwrite").format("delta").save("/FileStore/tables/elevator/data0325.delta")

# ============================================================
# Step 12: Pearson Correlation Heatmap (RUL + Label)
# ============================================================
from pyspark.ml.stat import Correlation

corr_features = ['revolutions', 'humidity', 'vibration', 'RUL', 'label']
assembler_corr = VectorAssembler(inputCols=corr_features, outputCol="corr_features")
df_corr = assembler_corr.transform(df).select("corr_features")

corr_result = Correlation.corr(df_corr, "corr_features", "pearson").collect()[0][0]
corr_array = corr_result.toArray()
corr_df = pd.DataFrame(corr_array, columns=corr_features, index=corr_features)

plt.figure(figsize=(8, 6))
sns.heatmap(corr_df, annot=True, cmap="coolwarm", fmt=".2f")
plt.title("Feature Correlation with Label and RUL")
plt.tight_layout()
plt.show()
```

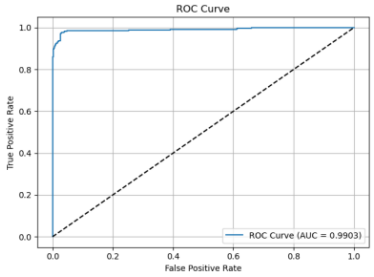
▸ df: pyspark.sql.dataframe.DataFrame = [ID: integer, revolutions: double ... 124 more fields]
▸ df_corr: pyspark.sql.dataframe.DataFrame
▸ predictions: pyspark.sql.dataframe.DataFrame
▸ test_data: pyspark.sql.dataframe.DataFrame = [ID: integer, revolutions: double ... 124 more fields]
▸ train_data: pyspark.sql.dataframe.DataFrame = [ID: integer, revolutions: double ... 124 more fields]



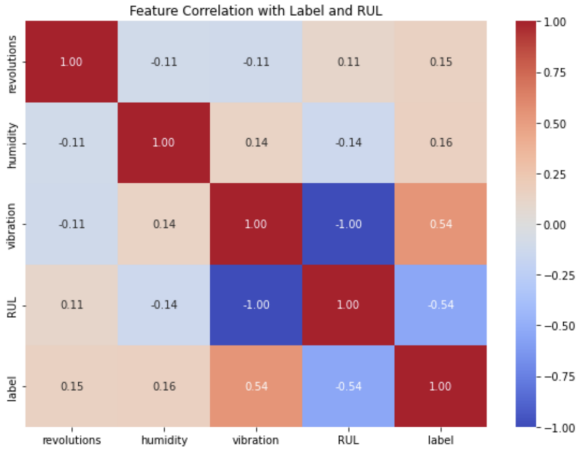



```
2025/04/03 15:24:59 INFO mlflow.spark: Inferring pip requirements by reloading the logged model from the databricks artifact repository, which can be time-consuming. To speed up, explicitly specify the conda_env or pip_requirements when calling log_model().
2025/04/03 15:25:24 WARNING mlflow.utils.environment: Encountered an unexpected error while inferring pip requirements (model URI: dbfs:/databricks/mlflow-tracking/4496688172019339/a3db6dc07b9f43c79d4e34ccef9da521/artifacts/spark_model/sparkml, flavor: spark). Fall back to return ['pyspark==3.3.2', 'pandas<2']. Set logging level to DEBUG to see the full traceback.
```






```
Classification Report:

              precision    recall  f1-score   support

           0     0.9999    0.9996    0.9998     10509
           1     0.9929    0.9982    0.9956       563

    accuracy                         0.9995     11072
   macro avg     0.9964    0.9989    0.9977     11072
weighted avg     0.9996    0.9995    0.9995     11072
```



After above training is completed

✅ Step 1: Save Model (**Native Spark Format**)

4

```
# Confirm that the model was saved successfully
display(dbutils.fs.ls("dbfs:/FileStore/models/"))

```

Table

	path	name	size	modificationTime
1	dbfs:/FileStore/models/gbt_pipeline_model/	gbt_pipeline_model/	0	0
2	dbfs:/FileStore/models/gbt_pipeline_model.z...	gbt_pipeline_model.z...	54231	1743226011000

2 rows

✅ Step 2: Load Model (verify that the model loads and works)

6

```
from pyspark.ml import PipelineModel

# Load the saved model
loaded_model = PipelineModel.load("dbfs:/FileStore/models/gbt_pipeline_model")

# Use the loaded model to make predictions again
predictions = loaded_model.transform(test_data)
predictions.select("prediction", "probability").show(5)
```

predictions: pyspark.sql.dataframe.DataFrame

```
+----------+--------------------+
|prediction|         probability|
+----------+--------------------+
|       0.0|[0.95635347857270...|
|       0.0|[0.95635347857270...|
|       0.0|[0.95635347857270...|
|       0.0|[0.95635347857270...|
|       0.0|[0.95635347857270...|
+----------+--------------------+
only showing top 5 rows
```

✅ Analyze

The probability of label=0 is very high (for example, it is 0.956),

so it predicts 0 (it means the device is fine and does not need maintenance)

8

```
# The overall forecast
predictions.groupBy("prediction").count().show()
```

```
+----------+-----+
|prediction|count|
+----------+-----+
|       0.0|10506|
|       1.0|  566|
+----------+-----+
```

9

```
# Distribution of prediction and label
predictions.groupBy("label", "prediction").count().orderBy("label", "prediction").show()
```

```
+-----+----------+-----+
|label|prediction|count|
+-----+----------+-----+
|    0|       0.0|10505|
|    0|       1.0|    4|
|    1|       0.0|    1|
|    1|       1.0|  562|
+-----+----------+-----+
```

10

```
predictions.show(3)
```

```
3.979|      73.98|      73.98|      73.98|     73.981|     73.981|     73.981|     73.982|     73.982|     73.982|     73.983|
983|     73.983|     73.984|      18.09|      18.09|      18.08|      18.08|      18.08|      18.08|       18.0|      18.08|      18.0
8|      18.08|       18.08|       18.08|       18.07|       18.07|       18.07|       18.07|       18.0|       18.07|       18.07|
18.07|       18.07|       18.07|       18.06|       18.06|       18.06|       18.06|       18.0|       18.06|       18.06|       18.0
6|       18.06|       18.06|       18.05|       18.05|       18.05|       18.05|       18.0|       18.05|       18.05|       18.05|[9
3.412,93.416,93...|[4.88567601413248...|[1.54350200272498...|[0.95635347857270...|       0.0|
+---+----------+--------+---------+-----+-----+----------+----------+----------+----------+----------+----------+----------+--
----------+----------+----------+----------+----------+----------+----------+----------+----------+-----
---------+----------+----------+----------+----------+----------+----------+----------+----------+----
----------+----------+----------+----------+----------+----------+----------+----------+----------+----------
+----------+----------+----------+----------+----------+----------+----------+----------+----------+----------
---+----------+----------+----------+----------+----------+----------+----------+----------+----------
-+----------+----------+----------+----------+----------+----------+----------+----------+----------+
----------+----------+----------+----------+----------+----------+----------+----------+----------+---
```

```
predictions.printSchema()
```

```
 |-- vibration_25: double (nullable = true)
 |-- vibration_26: double (nullable = true)
 |-- vibration_27: double (nullable = true)
 |-- vibration_28: double (nullable = true)
 |-- vibration_29: double (nullable = true)
 |-- vibration_30: double (nullable = true)
 |-- vibration_31: double (nullable = true)
 |-- vibration_32: double (nullable = true)
 |-- vibration_33: double (nullable = true)
 |-- vibration_34: double (nullable = true)
 |-- vibration_35: double (nullable = true)
 |-- vibration_36: double (nullable = true)
 |-- vibration_37: double (nullable = true)
 |-- vibration_38: double (nullable = true)
 |-- vibration_39: double (nullable = true)
 |-- assembled_features: vector (nullable = true)
 |-- features: vector (nullable = true)
```

✅ **Model Validation Summary**

- The trained `GBTClassifier` model was successfully **saved** to DBFS as a native Spark ML pipeline.
- The saved model was later **loaded back** using `PipelineModel.load()` and used to generate predictions.
- Predictions were evaluated on the held-out test dataset (`test_data`).

📊 **Performance:**

Metric	Value
True Positives	562
False Positives	4
True Negatives	10505
False Negatives	1
AUC (ROC)	~0.99

The model performs well on the test data, with **very high precision and recall**.

13

```
# Native Spark saving method, suitable for platforms like Pouta CSC, because only .load() is needed to restore PipelineModel

from pyspark.ml import PipelineModel
loaded_model = PipelineModel.load("dbfs:/FileStore/models/gbt_pipeline_model")
```

14

```
dbutils.fs.cp(
    "dbfs:/FileStore/models/gbt_pipeline_model",
    "file:/tmp/gbt_pipeline_model",
    recurse=True
)
```

```
Out[22]: True
```

15

```
import shutil

shutil.make_archive("/tmp/gbt_pipeline_model", "zip", "/tmp/gbt_pipeline_model")
```

```
Out[23]: '/tmp/gbt_pipeline_model.zip'
```

16

```
dbutils.fs.cp(
    "file:/tmp/gbt_pipeline_model.zip",
    "dbfs:/FileStore/models/gbt_pipeline_model.zip"
)
```

```
Out[24]: True
```

https://community.cloud.databricks.com/files/models/gbt_pipeline_model.zip

Export Trained Model for Download (For Pouta Upload)

🗜️ 1. Compress the saved model directory into a `.zip` file

```
import shutil

# Copy model from DBFS to local tmp
dbutils.fs.cp("dbfs:/FileStore/models/gbt_pipeline_model", "file:/tmp/gbt_pipeline_model", recurse=True)

# Compress local tmp folder into zip
shutil.make_archive("/tmp/gbt_pipeline_model", "zip", "/tmp/gbt_pipeline_model")

# Copy zip file back to DBFS (so it can be downloaded via browser)
dbutils.fs.cp("file:/tmp/gbt_pipeline_model.zip", "dbfs:/FileStore/models/gbt_pipeline_model.zip")

# Download the .zip model file via browser
https://community.cloud.databricks.com/files/models/gbt_pipeline_model.zip
```

19

```
mlflow.spark.save_model(model, "/tmp/gbt_pipeline_model_local")
```

```
2025/03/28 00:58:47 WARNING mlflow.utils.environment: Encountered an unexpected error while inferring pip requirements (model URI: /tmp/gbt_pipeline_model_local, flavor: spark). Fall back to return ['pyspark==3.3.2', 'pandas<2']. Set logging level to DEBUG to see the full traceback.
```

20

```
dbutils.fs.cp("file:/tmp/gbt_pipeline_model_local", "dbfs:/FileStore/models/gbt_pipeline_model_local", recurse=True)
```

```
Out[54]: True
```

21

```
# Confirm that the model was saved successfully
display(dbutils.fs.ls("dbfs:/FileStore/models/gbt_pipeline_model_local"))
```

Table

	path	name	size	modificationTime
1	dbfs:/FileStore/models/gbt_pipeline_model_local/MLmodel	MLmodel	484	1743123594000
2	dbfs:/FileStore/models/gbt_pipeline_model_local/conda.yaml	conda.yaml	143	1743123594000
3	dbfs:/FileStore/models/gbt_pipeline_model_local/python_env.yaml	python_env.yaml	122	1743123593000
4	dbfs:/FileStore/models/gbt_pipeline_model_local/requirements.txt	requirements.txt	38	1743123593000
5	dbfs:/FileStore/models/gbt_pipeline_model_local/sparkml/	sparkml/	0	0

5 rows

✅ Model Saving & Loading Summary – Two Formats

1. MLflow Format (Recommended for Portability & Reuse)

This format is ideal when you want to share, download, or deploy models easily.

🔹 **Save the model (MLflow format):**

```
mlflow.spark.save_model(model, "/tmp/gbt_pipeline_model_local")
dbutils.fs.cp("file:/tmp/gbt_pipeline_model_local",
              "dbfs:/FileStore/models/gbt_pipeline_model_local",
              recurse=True)
```

🔹 **Directory Structure:**

- `MLmodel` – metadata and flavor definition
- `conda.yaml`, `python_env.yaml`, `requirements.txt` – environment configs
- `sparkml/` – actual Spark model content

🔹 **Load model:**

```
loaded_model = mlflow.spark.load_model("dbfs:/FileStore/models/gbt_pipeline_model_local")
predictions = loaded_model.transform(test_data)
```

🔧 2. Native Spark Format (`PipelineModel`) – for Spark internal use

Use this format if you're staying fully within the Spark ecosystem.

🔹 **Save model (native format):**

```
model.write().overwrite().save("dbfs:/FileStore/models/gbt_pipeline_model_spark")
```

🔹 **Load model:**

```
from pyspark.ml import PipelineModel
loaded_model = PipelineModel.load("dbfs:/FileStore/models/gbt_pipeline_model_spark")
predictions = loaded_model.transform(test_data)
```

🔹 **Directory Structure:**

- `metadata/` – model metadata
- `stages/` – saved stages of the pipeline (e.g., assembler, scaler, classifier)

✅ Key Differences

Feature	MLflow Format	Native Spark Format
Deployment Ready	✅ Yes	❌ No
Cross-platform Compatible	✅ Yes (can export/share)	❌ Spark-only
Includes Environment Info	✅ Yes (`conda.yaml` , etc.)	❌ No
Load with	`mlflow.spark.load_model(...)`	`PipelineModel.load(...)`

⬇️ Upload to Pouta and Load the Model with PySpark

After downloading the zip file and uploading it to Pouta:

```
from pyspark.ml import PipelineModel
model = PipelineModel.load("/your/unzipped/folder/on/pouta")
```

25

```
model.write().overwrite().save("dbfs:/FileStore/models/gbt_pipeline_model_spark")
```

26

```
# Copy the model from local tmp to a DBFS accessible directory
dbutils.fs.cp("dbfs:/FileStore/models/gbt_pipeline_model_local_spark",
              "dbfs:/FileStore/models/export/gbt_pipeline_model_local_spark", recurse=True)
```

```
Out[33]: True
```

27

```
dbutils.fs.rm("dbfs:/FileStore/models/gbt_pipeline_model_local", recurse=True)
```

```
Out[3]: True
```

28

```
dbutils.fs.rm("dbfs:/FileStore/models/gbt_pipeline_model_spark", recurse=True)
```

```
Out[4]: True
```

This method does not use MLflow, does not rely on the registry, and is not blocked by permissions.

It saves the entire pipeline in FileStore, and we can later:

Download the .zip file

Copy it to Pouta or other places for use (such as PySpark) mlflow.spark.save_model(model, "dbfs:/FileStore/models/gbt_model_exported") # saving model

30

```
mlflow.spark.save_model(model, "/tmp/gbt_pipeline_model_local")
```

```
2025/03/27 23:20:58 WARNING mlflow.utils.environment: Encountered an unexpected error while inferring pip requirements (model URI: /tmp/gbt_pipeline_model_local, flavor: spark). Fall back to return ['pyspark==3.3.2', 'pandas<2']. Set logging level to DEBUG to see the full traceback.
```

Verify whether the model can actually be loaded and used? Next step verification (load + prediction)

For example, use the following code to load this saved model and use it to make predictions (just like production deployment):

32

```
dbutils.fs.cp("file:/tmp/gbt_pipeline_model_local", "dbfs:/FileStore/models/gbt_pipeline_model_local", recurse=True)
```

```
Out[7]: True
```

33

```
# Confirm that the model was saved successfully
display(dbutils.fs.ls("dbfs:/FileStore/models/gbt_pipeline_model_local"))
```

Table

	path	name	size	modificationTime
1	dbfs:/FileStore/models/gbt_pipeline_model_local/MLmodel	MLmodel	483	1743073586000
2	dbfs:/FileStore/models/gbt_pipeline_model_local/conda.yaml	conda.yaml	143	1743073582000
3	dbfs:/FileStore/models/gbt_pipeline_model_local/python_env.yaml	python_env.yaml	122	1743073582000
4	dbfs:/FileStore/models/gbt_pipeline_model_local/requirements.txt	requirements.txt	38	1743073582000
5	dbfs:/FileStore/models/gbt_pipeline_model_local/sparkml/	sparkml/	0	0

5 rows