Table of Contents

A. Research Question

A1. Research Question

The main research question of this study is:

Can we accurately predict wire-cutting machine failures using historical sensor data, employing Recurrent Neural Networks (RNN) enhanced with Long Short-Term Memory (LSTM) techniques?

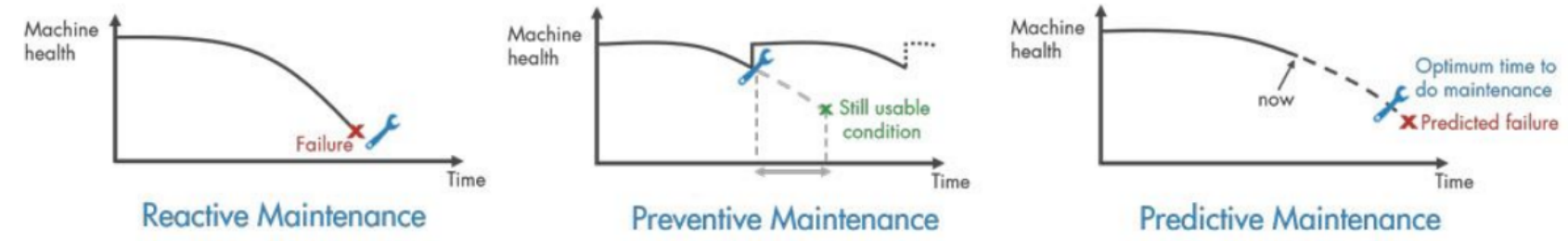
A2-1. Justification

By using predictive maintenance techniques, machine failures can be detected before they occur. This proactive approach helps avoid downtime and the associated costs (Mobley,2002). In our company, wire-cutting machines play a crucial role in our operation. The electrical components rely on the wires produced by these machines. Any malfunction can result in significant production delays and financial losses. Although the existing preventative maintenance procedure keeps the machines running, it does not maximize the equipment's remaining useful life (RUL) or cycles. Therefore, an accurate predictive model for machine failures can greatly improve operational efficiency and cost-effectiveness (Sikorska, Hodkiewicz & Ma , 2011).

To justify why this method of maintenance is important, let's discuss the different types of machinery maintenance. Reactive maintenance, also known as breakdown or "run-to-failure" maintenance, involves using equipment until a component fails (Sullivan et al., 2010). Only then are repairs or replacements carried out. While this approach may seem cost-saving, it can lead to unexpected downtime, which can ultimately be more expensive in the long run due to lost production time and possible damage to other components.

Preventive maintenance is another method where maintenance tasks are performed while the equipment is still operational to prevent unexpected breakdowns(Sullivan et al., 2010). These tasks are scheduled based on time (e.g., every month) or usage (e.g., after 1000 hours of operation). Preventive maintenance can help extend equipment life and prevent expensive repairs, but it also means the equipment may be serviced more often than necessary.

Here's the illustration of various types of maintanance:



Three types of maintenance and their approach to managing failures.

Source:https://www.mathworks.com/discovery/predictive-maintenance-matlab.html?s_tid=srchtitle_site_search_3_predictive%20maintenance

A2-2. Context

Predictive maintenance refers to the use of data-driven, proactive maintenance methods that are designed to predict when equipment failure might occur (Lee et al., 2014). The goal of predictive maintenance is to allow convenient scheduling of corrective maintenance and to prevent unexpected equipment failures. This approach can save money and time, reduce downtime, extend machinery life, and improve the safety and quality of operations (Mobley, 2002).

Recurrent Neural Networks (RNNs) are a type of artificial neural network designed to recognize patterns in sequences of data, such as text, genomes, handwriting, or the time series data used in predictive maintenance (Lipton et al., 2015). Long Short-Term Memory (LSTM) networks, a special type of RNN, were developed to combat the problem of vanishing gradients that makes it difficult for the network to learn and tune the parameters for long sequences (Hochreiter & Schmidhuber, 1997). LSTM networks, have been especially effective in predictive maintenance because they can process and remember patterns in data collected over time (Malhotra et al., 2016).

Previous studies have shown that LSTM networks can effectively predict failures in industrial machinery using data from sensors installed on the machines (Zhao et al., 2019). In this study, the goal is to employ the same techniques to predict when wire-cutting machines might fail, aiming to improve the machines' performance and reduce the cost of maintenance.

A3. Hypothesis

Null Hypothesis (H0): There is no statistically significant relationship between the historical sensor data and wire-cutting machine failures.

Alternate Hypothesis (H1): There is a statistically significant relationship between the historical sensor data and wire-cutting machine failures.

The hypotheses are based on the premise that the functioning status of each machine (normal operation vs. failure) is reflected in its sensor readings, and by learning from historical data, it is possible to predict future machine failures.

Utilizing RNNs that employ LSTMs can assist in modeling the correlation between sensor readings and machine breakdowns over time by capturing temporal dependencies in the data (Hochreiter & Schmidhuber, 1997).

B. Data Collection

B1. Data Collection Process

The dataset in this analysis is a simulation of multivariate time series data, representing sensor readings from industrial machinery units. Expertise in engineering was employed to generate this data, ensuring a realistic representation of sensor streams over a 10-month period. The data was acquired from the UCI Machine Learning Repository, a recognized resource for open datasets used extensively in data science education and research.

B2. Relevant Data Description

The dataset encompasses several key variables:

1. `Date` : The date of the sensor reading.
2. `Device` : The ID of the wire-cutting machine.
3. `Failure` : Indicates whether the machine failed on this day (1 signifies machine failure, 0 otherwise).
4. `Metric1` to `Metric9` : These are sensor readings from the machine. The specific nature of these metrics is not disclosed in the dataset, maintaining the anonymity of the manufacturing process. The dataset contains `124,494` rows and `12` columns. This is a large dataset, which is needed for the model's performance as deep learning models like RNN-LSTM generally perform better with more data.

B3. Advantages and Disadvantages of Data-Gathering Methodology

A major advantage of using synthetically generated data is the ability to develop and evaluate models without requiring access to actual proprietary operational data from real sensor streams which can be difficult to obtain in practice (Dilmegani, 2022). However, the disadvantage of using synthetic data is that it lacks accuracy and may not capture all possible failure modes and scenarios that could occur in actual wire-cutting machines (Anand, 2022).

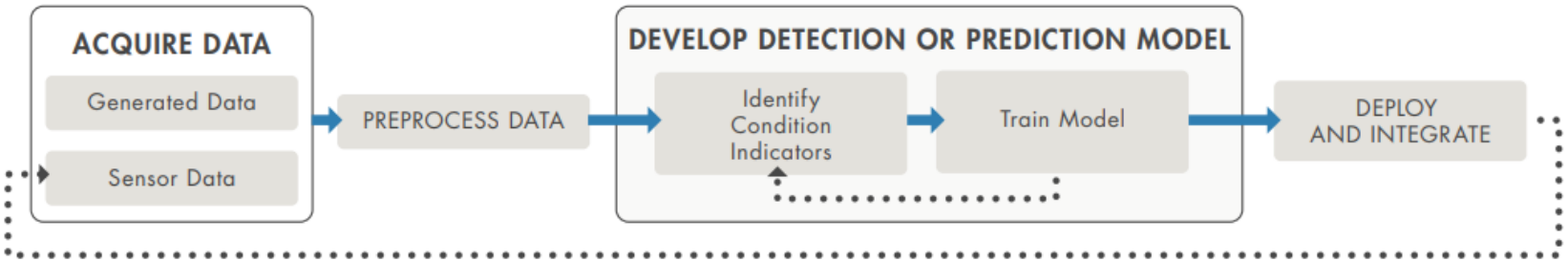
B4. Overcoming Data Collection Challenges

The dataset used in this study had some challenges. It was made up of sensor readings (metrics) from hypothetical machines, but these readings weren't clearly labeled or described. This made it hard to know what each reading meant. Nevertheless, for building a model to predict machine failures, knowing the exact meaning of each reading isn't always necessary. Instead, it's important to look for patterns in the readings that might signal

a machine is about to fail. So, these readings were treated as general numbers and used to train the model. Another challenge was that failures were much less common than non-failures in the dataset (imbalanced dataset). This is common in maintenance data because machines usually work fine and only fail occasionally. But, if this imbalance is not addressed, it can lead to a model that is good at predicting non-failures but bad at predicting failures (Jayaswal, 2020). To fix this, the suggestion was to balance the data by either increasing the number of failure cases(oversampling) or decreasing the number of non-failure cases (undersampling). Also, using measures that consider both failures and non-failures, like precision, recall, F1 score, and AUCROC, were recommended (Mwiti, 2020). These steps helped to prepare the data for analysis and modeling, making the predictions from the model more reliable.

C. Data Extraction and Preparation

Overview of Predictive Maintance Workflow



The basics of the predictive maintenance workflow.

Source:https://www.mathworks.com/content/dam/mathworks/white-paper/gated/predictive-maintenance-challenges-whitepaper.pdf

C1-1. Data Extraction

In the data extraction stage, the dataset is loaded from a CSV file into a Pandas DataFrame. This format is convenient for data analysis and manipulation in Python.

Here are the steps to extract the data

1. Import neccessary libraries
2. Load the dataset points.

The code below will show how these steps are performed.

In [2]:
```
# Importing the required libaries
import os
import cv2
```

```
from PIL import Image
import numpy as np
import pandas as pd
# from patchify import patchify
from sklearn.preprocessing import MinMaxScaler,StandardScaler
import matplotlib.pyplot as plt
import seaborn as sns
import random
from tensorflow import keras
import keras
from keras.models import Model
from keras.layers import Input, Conv2D, GlobalMaxPool2D, UpSampling2D, Conv2DTranspose, concatenate, BatchNormalization, Dropout, Lam
from keras import backend as K
from sklearn.model_selection import train_test_split
from tensorflow.keras.callbacks import EarlyStopping
from imblearn.over_sampling import SMOTE
from tensorflow.keras.layers import Dense, LSTM, Dropout, Input, Flatten, Attention
from tensorflow.keras.models import Model
from tensorflow.keras.regularizers import l2
from imblearn.under_sampling import RandomUnderSampler
from imblearn.over_sampling import ADASYN
from tensorflow.keras.layers import SimpleRNN
```

In [3]:
```
# Let's first unzip the file to access its contents.
import zipfile

# Specify the location of the zip file
zip_path = "/content/drive/MyDrive/archive.zip"

# Specify the path where the file will be extracted
extract_path = "/content/drive/MyDrive/"

# Create a ZipFile object
with zipfile.ZipFile(zip_path, 'r') as zip_ref:
    # Extract all the contents of the zip file into the specified directory
    zip_ref.extractall(extract_path)
```

In [4]:
```
# Load the CSV data into a pandas DataFrame
data_path = extract_path + 'predictive_maintenance_dataset.csv'
data = pd.read_csv(data_path)

# Display the first few rows of the DataFrame
data.head()
```

Out[4]:

	date	device	failure	metric1	metric2	metric3	metric4	metric5	metric6	metric7	metric8	metric9
0	1/1/2015	S1F01085	0	215630672	55	0	52	6	407438	0	0	7
1	1/1/2015	S1F0166B	0	61370680	0	3	0	6	403174	0	0	0
2	1/1/2015	S1F01E6Y	0	173295968	0	0	0	12	237394	0	0	0
3	1/1/2015	S1F01JE0	0	79694024	0	0	0	6	410186	0	0	0
4	1/1/2015	S1F01R2B	0	135970480	0	0	0	15	313173	0	0	3

In [5]:
```
# Move the target column to the end
cols = list(data.columns)
cols.remove('failure')
cols.append('failure')
data = data[cols]

# Display the first few rows of the reordered DataFrame
data.head()
```

Out[5]:

	date	device	metric1	metric2	metric3	metric4	metric5	metric6	metric7	metric8	metric9	failure
0	1/1/2015	S1F01085	215630672	55	0	52	6	407438	0	0	7	0
1	1/1/2015	S1F0166B	61370680	0	3	0	6	403174	0	0	0	0
2	1/1/2015	S1F01E6Y	173295968	0	0	0	12	237394	0	0	0	0
3	1/1/2015	S1F01JE0	79694024	0	0	0	6	410186	0	0	0	0
4	1/1/2015	S1F01R2B	135970480	0	0	0	15	313173	0	0	3	0

In [6]:
```
# Check the size of the dataset
dataset_size = data.shape
print("The size of the dataset is:", dataset_size)
print('----------------'*5)

# Get a summary of the data
data_summary = data.describe()
print('Statistical summary of the dataset\n','----------------'*5,'\n')
data_summary
# print('----------------'*5)
# print('----------------'*5)
# Check the distribution of the failure variable
# failure_distribution = data['failure'].value_counts()
# print("Distribution of the failure variable:\n", failure_distribution)

# dataset_size, data_summary, missing_values, failure_distribution
```

```
The size of the dataset is: (124494, 12)
------------------------------------------------------------------------------
Statistical summary of the dataset
 ------------------------------------------------------------------------------
```

Out[6]:

	metric1	metric2	metric3	metric4	metric5	metric6	metric7	metric8	metric9	failure
count	1.244940e+05	124494.000000	124494.000000	124494.000000	124494.000000	124494.000000	124494.000000	124494.000000	124494.000000	124494.000000
mean	1.223881e+08	159.492706	9.940897	1.741120	14.222669	260172.657726	0.292528	0.292528	13.013848	0.000851
std	7.045933e+07	2179.677781	185.748131	22.908507	15.943028	99151.078547	7.436924	7.436924	275.661220	0.029167
min	0.000000e+00	0.000000	0.000000	0.000000	1.000000	8.000000	0.000000	0.000000	0.000000	0.000000
25%	6.128476e+07	0.000000	0.000000	0.000000	8.000000	221452.000000	0.000000	0.000000	0.000000	0.000000
50%	1.227974e+08	0.000000	0.000000	0.000000	10.000000	249799.500000	0.000000	0.000000	0.000000	0.000000
75%	1.833096e+08	0.000000	0.000000	0.000000	12.000000	310266.000000	0.000000	0.000000	0.000000	0.000000
max	2.441405e+08	64968.000000	24929.000000	1666.000000	98.000000	689161.000000	832.000000	832.000000	70000.000000	1.000000

The dataset contains 124,494 rows and 12 columns. Each row represents a unique sensor reading from a specific machine on a specific date. The columns represent the date of the reading, the ID of the machine (device), a failure indicator (failure), and nine different metrics presumably representing different sensor readings (metric1 through metric9). The failure indicator is a binary variable, with 1 indicating that the machine failed on that day, and 0 indicating that it did not.

The failure column, which is our target variable for prediction, is highly imbalanced. Out of 124,494 observations, only 106 machines have failed which is significantly less compared to the 124,388 observations where no failure was reported. This class imbalance is common in predictive maintenance scenarios, where failures are typically rare events compared to normal operation.

C1-2. Data Preparation Process

The data preparation stage involves several steps aimed at making the dataset suitable for deep learning modeling. These steps are crucial as the quality of data and the usefulness of derived features directly influence the model's performance.

1. **Handling Missing Values**: In the initial data exploration, no missing values were found in the dataset. If there were missing values, different missing-value-handling methods would have been used, whether by deleting the rows or columns with missing values or by imputation.

2. **Handling Outliers**: As observed in the boxplots below, some of the sensor readings contain outliers. These outliers can skew the model's learning and negatively affect its performance.

3. **Encoding Categorical Variables**: The `device` column, which represents the ID of the machine, is a categorical variable. Machine learning algorithms require input variables to be numeric. We will handle this by applying one-hot encoding or ordinal encoding.

4. **Normalizing Numerical Variables**: The sensor readings (`metric1` through `metric9`) are numerical variables with varying ranges. Many machine learning algorithms perform better when numerical input variables are scaled to a standard range. We will handle this by applying normalization or standardization.

5. **Handling Class Imbalance**: As observed in the bar chart, the `failure` column, which is our target variable for prediction, is highly imbalanced. This can bias the model's learning towards the majority class. We will handle this by applying a suitable method, such as oversampling the minority class, undersampling the majority class, or using a combination of both (SMOTE).

I. **Handling Missing Values**

Here's the code to show if there are missing values in the dataset

In [ ]:
```
# Check for missing values
missing_values = data.isnull().sum()
print('Checking for the missing values in the dataset:\n{}'.format( missing_values))
```

```
Checking for the missing values in the dataset:
date       0
device     0
metric1    0
metric2    0
metric3    0
metric4    0
metric5    0
metric6    0
metric7    0
metric8    0
metric9    0
failure    0
dtype: int64
```

In the initial data exploration, no missing values were found in the dataset. If there were missing values, different missing-value-handling methods would have been used, whether by deleting the rows or columns with missing values or by imputation.

In [7]:
```
# Check for duplicates
print(data.duplicated().sum())
```

```
1
```

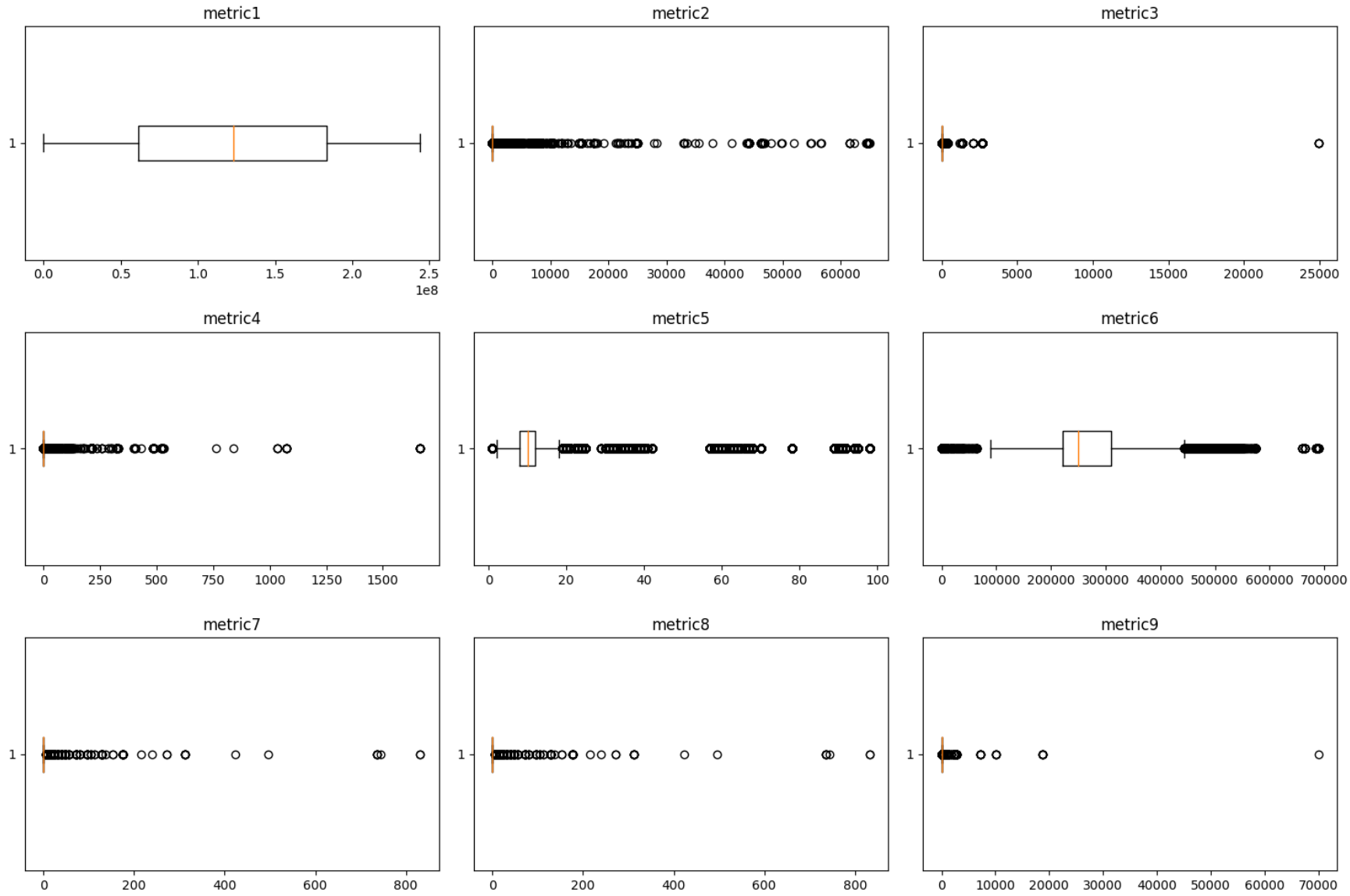
In [8]:
```
# Remove duplicate rows
data = data.drop_duplicates()
```

```
# Verify that duplicates have been removed
print("Number of duplicates after removal: ", data.duplicated().sum())
```

```
Number of duplicates after removal:  0
```

II. **Handling Outliers**

In [ ]:
```
# Import the necessary library
import matplotlib.pyplot as plt

# We will investigate the numerical variables (metrics) for outliers
metrics = [f'metric{i}' for i in range(1, 10)]

# Plot boxplots for each metric
plt.figure(figsize=(15, 10))
for i, metric in enumerate(metrics):
    plt.subplot(3, 3, i + 1)
    plt.boxplot(data[metric], vert=False)
    plt.title(metric)

plt.tight_layout()
plt.show()
```



The boxplots indicate the presence of outliers in several metrics, as shown by the points located outside the interquartile range (IQR) in each boxplot. But in the context of machine sensor data, outliers might represent significant events or abnormalities that could potentially indicate a machine failure (Chandola et al., 2009). As such, rather than removing these outliers, it may be beneficial to retain them in the dataset as they could provide valuable information for the predictive model (Hodge & Austin, 2004).

In [9]:
```
df = data.copy()
```

III. **Encoding Categorical Variables**

The `device` column, which represents the ID of the machine, is a categorical variable. Machine learning algorithms require input variables to be numeric. We will handle this by applying one-hot encoding or label encoding.

In [10]:
```
# Check the number of unique devices
num_devices = df['device'].nunique()
print(f'There are {num_devices} unique devices in the dataset.')

# Since there are a large number of unique devices, it's not practical to use one-hot encoding.
# Instead, we'll use label encoding, which assigns a unique number to each category.
from sklearn.preprocessing import LabelEncoder

encoder = LabelEncoder()
df['device'] = encoder.fit_transform(df['device'])

# Display the first few rows of the DataFrame to verify the encoding
df.head()
```

```
There are 1169 unique devices in the dataset.
```

Out[10]:

	date	device	metric1	metric2	metric3	metric4	metric5	metric6	metric7	metric8	metric9	failure
0	1/1/2015	0	215630672	55	0	52	6	407438	0	0	7	0
1	1/1/2015	2	61370680	0	3	0	6	403174	0	0	0	0
2	1/1/2015	3	173295968	0	0	0	12	237394	0	0	0	0
3	1/1/2015	4	79694024	0	0	0	6	410186	0	0	0	0
4	1/1/2015	5	135970480	0	0	0	15	313173	0	0	3	0

In [11]:
```
df['device'].values
```

Out[11]:
```
array([   0,    2,    3, ..., 1075, 1081, 1082])
```

The 'device' variable has been successfully encoded into numbers, allowing it to be used as an input for a RNN-LSTM model.

IV. **Normalizing Numerical Variables**

The sensor readings (`metric1` through `metric9`) are numerical variables with varying ranges. Many machine learning algorithms, including neural networks, often yield better results when numerical input variables are scaled to a standard range (Brownlee, 2020). This can be addressed by applying normalization or standardization techniques.

Next, normalization technique will be used. Normalization is a scaling technique where values are shifted and rescaled to a standard range, such as 0 to 1 or -1 to 1 (Bhandari, 2023). This technique is particularly beneficial for neural networks, which tend to perform better when input values are small and on a similar scale (Goodfellow et al., 2018). For this purpose, we'll use Min-Max normalization, which scales the data to the range of 0-1.

For our dataset, we'll normalize the numerical sensor metrics. It's important to note that we should not normalize the target variable 'failure', as it is a binary classification label.

In [ ]:
```
# Import the necessary library
from sklearn.preprocessing import MinMaxScaler
metrics = ['metric1', 'metric2', 'metric3', 'metric4', 'metric5',
       'metric6', 'metric7', 'metric8', 'metric9']
# Initialize a scaler
scaler = MinMaxScaler()

# Normalize the numerical columns
df[metrics] = scaler.fit_transform(df[metrics])

# Display the first few rows of the DataFrame to verify the normalization
df.head()
```

Out[ ]:

	date	device	metric1	metric2	metric3	metric4	metric5	metric6	metric7	metric8	metric9	failure
0	1/1/2015	0	0.883224	0.000847	0.00000	0.031212	0.051546	0.591204	0.0	0.0	0.000100	0
1	1/1/2015	2	0.251374	0.000000	0.00012	0.000000	0.051546	0.585017	0.0	0.0	0.000000	0
2	1/1/2015	3	0.709821	0.000000	0.00000	0.000000	0.113402	0.344461	0.0	0.0	0.000000	0
3	1/1/2015	4	0.326427	0.000000	0.00000	0.000000	0.051546	0.595191	0.0	0.0	0.000000	0
4	1/1/2015	5	0.556935	0.000000	0.00000	0.000000	0.144330	0.454420	0.0	0.0	0.000043	0

The numerical sensor metrics have been successfully normalized to the range 0-1, as confirmed by the values in the DataFrame.

V. **Handling Class Imbalance**

The 'failure' column, the target variable for prediction, is highly imbalanced, as observed below (124,387 non-failures and 106 failures). This imbalance can bias the predictive model's learning towards the majority class, which in this case is non-failures (Haibo et al., 2008). Addressing this imbalance is a crucial step in the preprocessing of our data.

We will handle the class imbalance by applying the ADASYN (Adaptive Synthetic) technique during the data preprocessing stage. The ADASYN technique generates synthetic data for the minority class (failures) based on the density distribution of the original data. This technique compensates for the imbalance by generating more synthetic data for instances that are harder to learn, rather than the ones that are easier.

This method helps in maintaining the temporal structure of the data while improving the balance of the classes. It also adjusts the model's perception of the importance of each class by increasing the representation of the minority class (Haibo et al., 2008).

The effectiveness of using ADASYN will be evaluated by monitoring the model's performance on the minority class during training. If the performance is found to be unsatisfactory, we will consider revisiting the decision and potentially applying other techniques to further address the imbalance.

In [ ]:
```
# Checking the distribution of failures
failure_distribution = data['failure'].value_counts()
print('The distribution of non-failures and failures:\n{} '.format(failure_distribution))
```

```
The distribution of non-failures and failures:
0    124387
1       106
Name: failure, dtype: int64
```

Failure Distribution: The vast majority of the records in the dataset are for functioning devices, with 124,387 instances of no failure (indicated by '0') and only 106 instances of failure (indicated by '1'). This shows that the dataset is highly imbalanced with respect to the failure variable. This imbalance is something we'll need to consider during the modeling phase.

C2. Tools and Techniques Used

Tools

The data extraction and preparation process is performed in Python using several libraries:

- Pandas: Used for data manipulation and analysis. It offers data structures and operations for manipulating numerical tables and time-series data (McKinney, 2022).
- NumPy: Used for handling numerical data. It provides support for arrays, along with a collection of mathematical functions to operate on these arrays (Oliphant, 2006).
- Scikit-learn: Used for data preprocessing and machine learning. It provides various tools for data mining and data analysis, including tools for handling missing values, encoding categorical variables, and normalizing numerical variables (Pedregosa et al., 2011).
- Imbalanced-learn: Used for handling class imbalance. It is a Python library offering various resampling techniques commonly used in datasets showing strong between-class imbalance (Lemaître et al., 2017).

Techniques

- Exploratory Data Analysis (EDA)
- Time-series analysis
- Descriptive statistics

- Data wrangling
- Handling Class Imbalance

C3. Justification for Tools and Techniques

The chosen tools and techniques are widely used in data science and machine learning projects due to their versatility, efficiency, and ease of use. Let's break down the explanation into two sections: tools and techniques.

Tools

Python, Pandas, Matplotlib, Seaborn, and Scikit-learn are widely used in data science and machine learning projects due to their versatility, efficiency, and ease of use (VanderPlas, 2023).

Techniques

Time-series analysis is crucial for datasets where temporal patterns, trends, or anomalies are important for predictions, such as in the case of predicting machine failures based on sensor data collected over time (Shumway & Stoffer, 2017).

Descriptive statistics provide a fundamental understanding of the data's distribution, which is crucial for selecting appropriate data preprocessing and modeling techniques (Hastie et al., 2017).

EDA is a vital initial step in data analysis that helps identify potential issues such as outliers or missing values. It uncovers underlying patterns and relationships in the data, providing valuable insights that can guide the subsequent analysis process (Tukey, 2020).

Data wrangling, which includes handling missing values, encoding categorical variables, and normalizing numerical variables, is essential for transforming raw data into a suitable format for machine learning models (Turing Enterprise, 2022).

Finally, handling class imbalance is a critical consideration in machine learning, especially for imbalanced datasets like the one in this study. Although no specific resampling technique was applied due to the nature of the data and the chosen model, it remains an important consideration to ensure that the model does not become biased towards the majority class (Lemaître et al., 2017).

C4. Advantages and Disadvantages of Data Extraction and Preparation Methods

Advantages:

- **Effectiveness**: The chosen techniques are effective in preparing the dataset for machine learning. They tackle common data issues such as missing values, outliers, categorical variables, varying ranges of numerical variables, and class imbalance.
- **Efficiency**: The chosen tools are efficient and versatile in processing large datasets and suitable for various data problems. They are optimized for performance and offer efficient implementations of common data operations and machine learning algorithms.

Disadvantages:

- **Assumptions**: The chosen techniques make certain assumptions about the data. For example, the method for handling outliers assumes that extreme values are errors, but they could be important signals in some cases (Brownlee, 2020). Similarly, the method for handling class imbalance assumes that the class distribution is important, but this might not be the case for some machine learning algorithms.
- **Complexity**: The data preparation process can be complex and time-consuming. It requires careful selection and application of methods, along with tuning of parameters. Improper data preparation can lead to poor model performance (Ghosh et al., 2021).

Exploratory Data Analysis

We will now conduct exploratory data analysis (EDA) on the dataset. EDA is an essential step in data analysis and machine learning because it aids in comprehending the variable distributions, their interrelationships, and identifying any possible outliers or errors in the data.

Descriptive Statistics

In [ ]:
```
# Generate descriptive statistics of the dataset
data.describe(include='all')
```

Out[ ]:

	date	device	metric1	metric2	metric3	metric4	metric5	metric6	metric7	metric8	
count	124493	124493	1.244930e+05	124493.000000	124493.000000	124493.000000	124493.000000	124493.000000	124493.000000	124493.000000	124493
unique	304	1169	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
top	1/1/2015	Z1F0QLC1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
freq	1163	304	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
mean	NaN	NaN	1.223875e+08	159.493988	9.940977	1.741134	14.222719	260173.031022	0.292531	0.292531	13
std	NaN	NaN	7.045934e+07	2179.686488	185.748875	22.908598	15.943082	99151.389285	7.436954	7.436954	275
min	NaN	NaN	0.000000e+00	0.000000	0.000000	0.000000	1.000000	8.000000	0.000000	0.000000	0
25%	NaN	NaN	6.128346e+07	0.000000	0.000000	0.000000	8.000000	221452.000000	0.000000	0.000000	0
50%	NaN	NaN	1.227971e+08	0.000000	0.000000	0.000000	10.000000	249800.000000	0.000000	0.000000	0
75%	NaN	NaN	1.833091e+08	0.000000	0.000000	0.000000	12.000000	310266.000000	0.000000	0.000000	0
max	NaN	NaN	2.441405e+08	64968.000000	24929.000000	1666.000000	98.000000	689161.000000	832.000000	832.000000	70000

The discussion of statistical summary of the dataset is as follows:

- **date**: The dataset spans 304 unique dates. The date with the highest frequency is '1/1/2015' with 1163 records.
- **device**: There are 1169 unique devices in the dataset. The device with the most records is **'Z1F0QLC1'** with 304 records.
- **failure**: This is the target variable, indicating whether a failure occurred (1) or not (0). The mean of this variable is approximately 0.000851, suggesting that failures are rare events. This is consistent with the fact that machine failures are typically infrequent occurrences.
- **metric1** to **metric9**: These are the sensor readings. The mean, standard deviation (std), minimum (min), 25th percentile (25%), median (50%), 75th percentile (75%), and maximum (max) values of these metrics vary widely. This suggests different scales and distributions for each metric. For instance, metric1 has a mean of approximately 1.22e+08 and a maximum of 2.44e+08, while metric2 has a mean of approximately 159.49 and a maximum of 64968.
- It's also worth noting that some metrics have a minimum of 0 and 25% percentile of 0, indicating that a significant portion of the readings for these metrics are 0. This could imply that these sensors were not active or not relevant all the time.

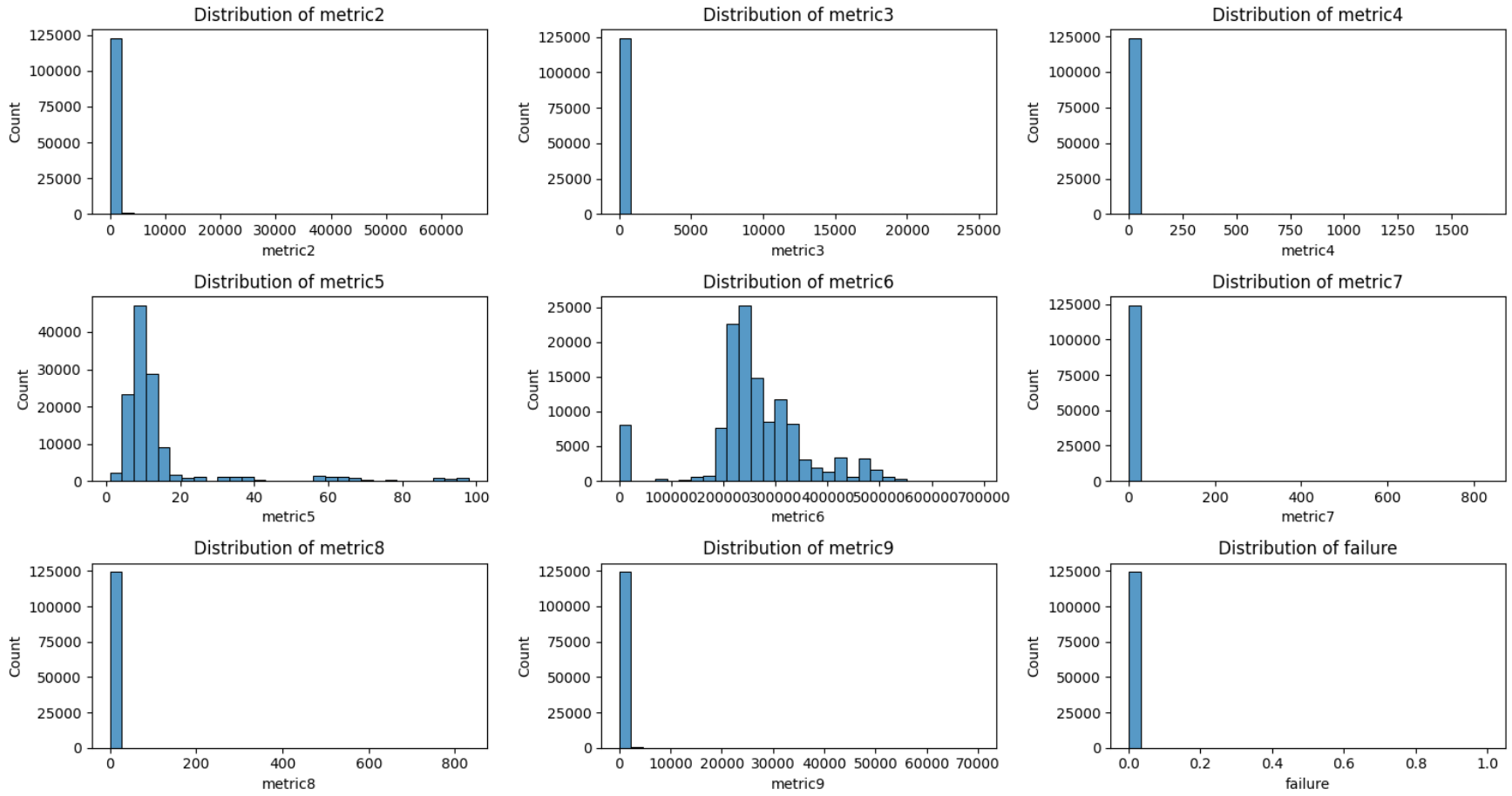
The standard deviations of the metrics indicate the variability in the sensor readings. Some metrics, such as metric2 and metric3, have a high standard deviation, indicating a high variability in the sensor readings. In contrast, other metrics have a low standard deviation, indicating that the sensor readings are more consistent.

The dataset does not appear to have any missing values, as the count is the same for all columns.

Univariate Analysis

We will start by examining the distribution of individual variables.We'll use histograms to visualize the distributions of the individual metrics.

In [ ]:
```
import matplotlib.pyplot as plt
import seaborn as sns

# Set up the matplotlib figure
f, axes = plt.subplots(3, 3, figsize=(15, 8))

# Generate a histogram for each metric
for i, metric in enumerate(data.columns[3:]):
    ax = axes[i//3, i%3]
    sns.histplot(data[metric], bins=30, ax=ax)
    ax.set_title(f"Distribution of {metric}")

# Adjust the layout
f.tight_layout()
```



The histograms provide insights into the distributions of the individual metrics:

- **metric1** has a fairly uniform distribution, with values spread out evenly across the range.
- **metric2**, **metric3**, and **metric4** have highly skewed distributions, with the majority of the values being close to zero.
- **metric5** and **metric6** show a multimodal distribution, with several peaks.
- **metric7**, **metric8**, and **metric9** have similar distributions as **metric2**, **metric3**, and **metric4**, with most values close to zero.

The histograms show that majority of the metrics are highly skewed, with most values concentrated at lower ranges. This skewness is not unusual in sensor data, especially for metrics that measure rare events. However, it's worth noting that certain machine learning models, like linear regression, assume that the input variables are normally distributed (Bobbitt, 2021). This assumption does not hold for our data.

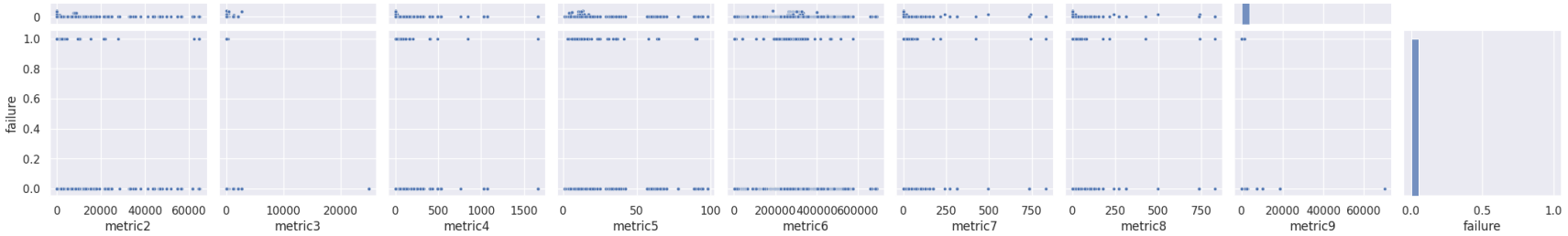
For the LSTM model we are planning to use, this assumption is not required, so we do not need to transform these variables to be normally distributed. Regardless, it is still beneficial to normalize the variables so that they are on the same scale. This can help the LSTM model to train more efficiently.

Bivariate Analysis

We will examine the relationships between pairs of variables. A pairplot or a scatterplot matrix can help us visualize these relationships.

In [ ]:
```
# Set the size of the plot
sns.set(rc={'figure.figsize':(6, 6)})

# Create a pairplot of the subset of the data, fitting it into the screen
sns.pairplot(data.iloc[:, 3:], corner=True, plot_kws={"s": 10})

# Display the plot
plt.show()
```


metric3
25000
20000
15000
10000
5000
0
metric4
1500
1000
500
0
metric5
100
80
60
40
20
0
metric6
600000
400000
200000
0
metric7
800
600
400
200
0
metric8
800
600
400
200
0
metric9
60000
40000
20000



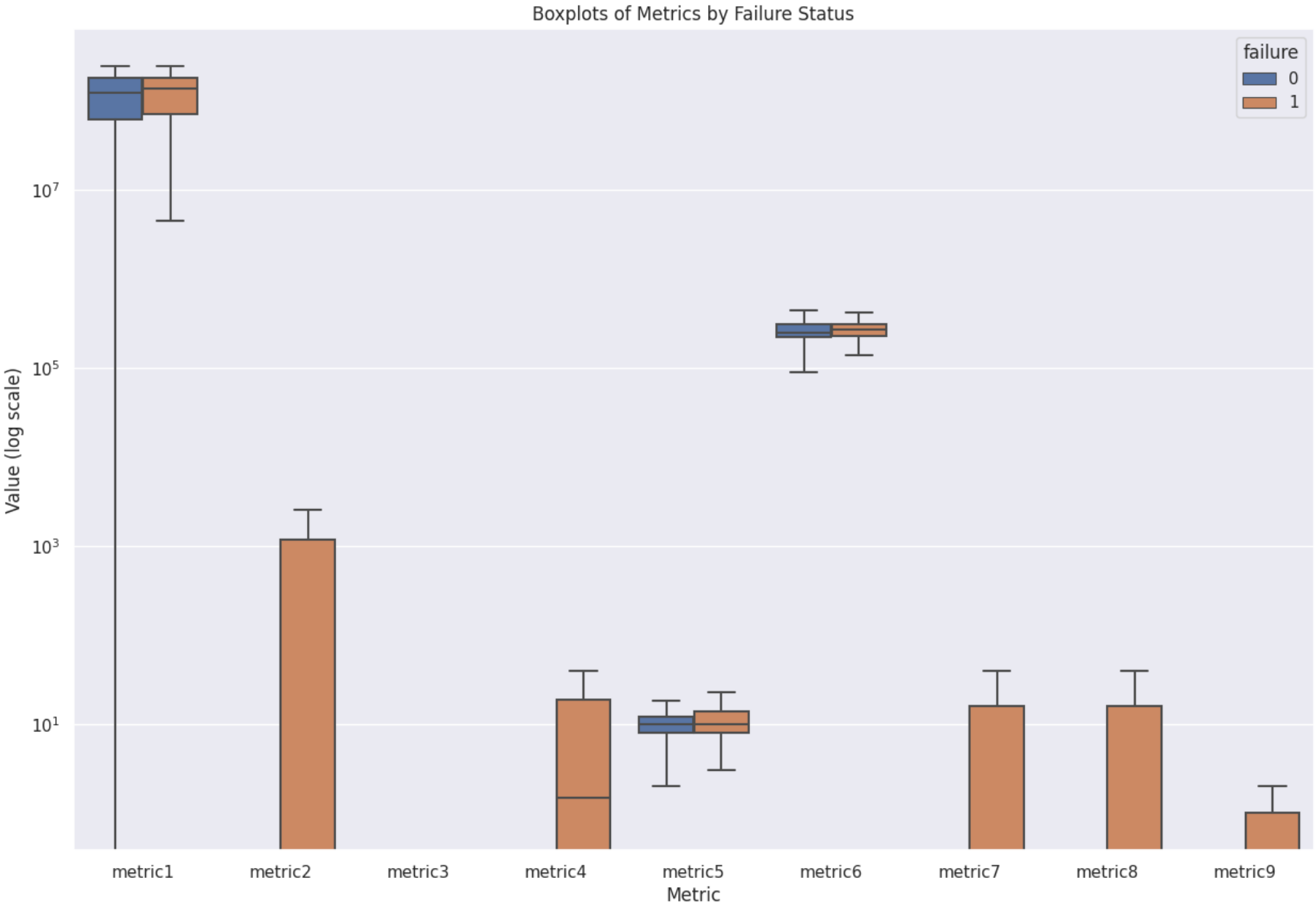
The pairplot provides insights into the relationships between pairs of metrics. Here are some key observations:

- Most of the metrics show no obvious correlation, as indicated by the lack of any clear patterns in the scatterplots. This suggests that these metrics are largely independent of each other.
- `Metric7` and `Metric8` show a perfect positive linear relationship, as indicated by the diagonal line in their scatterplot. This is also confirmed by the correlation matrix in the next analysis.
- Most of the scatterplots show a large concentration of points near zero, reflecting the skewed distributions we observed in the histograms.

Bivariant Analysis of Failure Status and Variables

We'll examine the relationship between the metrics and the failure status. We'll use boxplots to compare the distributions of each metric on days with failures and days without failures.

In [ ]:
```
# Import the necessary library
import matplotlib.pyplot as plt
import seaborn as sns

# Melt the data for the boxplots
melted_data = pd.melt(data, id_vars=['failure'], value_vars=data.columns[2:-1])

# Plot the boxplots
plt.figure(figsize=(15, 10))
sns.boxplot(x='variable', y='value', hue='failure', data=melted_data, showfliers=False)
plt.yscale('log')
plt.title('Boxplots of Metrics by Failure Status')
plt.ylabel('Value (log scale)')
plt.xlabel('Metric')
plt.show()
```



The boxplots offer some insights into how the distributions of the metrics differ on days with failures versus days without failures:

1. For most metrics (`metric1` , `metric2` , `metric3` , `metric4` , `metric5` , `metric6`), the median value on days with failures is not notably different from the median value on days without failures. This suggests that these metrics on their own might not be highly predictive of machine failure.

2. It appears that `metric7`, `metric8`, and `metric9` have higher median values on days when failures occur compared to days when there are no failures. This observation indicates that these metrics could be crucial in predicting failures.

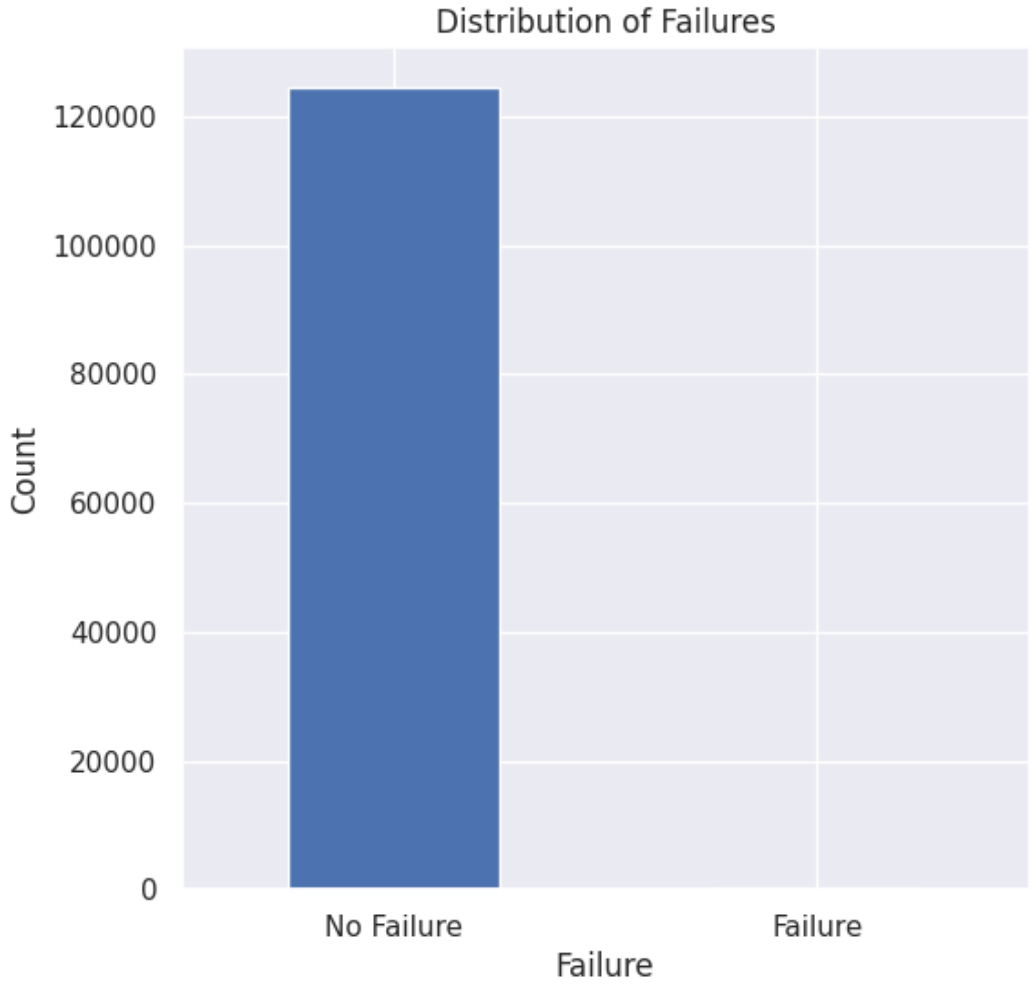
3. The interquartile ranges (the height of the boxes) are typically larger on days with failures, indicating greater variability in the metrics on these days.

4. There seem to be many outliers (values that fall outside the range of the box and whiskers) in the metrics on both failure and non-failure days. Some of these outliers could potentially be due to errors or anomalies in the data, while others may reflect genuine variability in the metrics.

5. The boxplots confirm the strong skewness in the distribution of the metrics, with most values being close to zero and a long tail of higher values. This is why we used a logarithmic scale for the y-axis.

Analyzing Machine Failures

In [ ]:
```
# First, let's look at the distribution of failures in the dataset
failure_counts = data['failure'].value_counts()
failure_counts.plot(kind='bar')
plt.xlabel('Failure')
plt.ylabel('Count')
plt.title('Distribution of Failures')
plt.xticks([0,1], ['No Failure', 'Failure'], rotation=0)
plt.show()
```



The failure column, which is our target variable for prediction, is highly imbalanced. Out of 124,494 observations, only 106 machines have failed which is significantly less compared to the 124,388 observations where no failure was reported. This class imbalance is common in predictive maintenance scenarios, where failures are typically rare events compared to normal operation.

In [ ]:
```
# Machines that failed at least once
failed_machines = data[data['failure'] == 1]['device'].unique()
num_failed_machines = len(failed_machines)
print("Total number of failed machines:",num_failed_machines)
print('\nThe following are the machines that failed at least once:\n',failed_machines)
```

```
Total number of failed machines: 106

The following are the machines that failed at least once:
 ['S1F0RRB1' 'S1F0CTDN' 'W1F0PNA5' 'W1F13SRV' 'W1F1230J' 'W1F0T034'
 'S1F0GG8X' 'S1F023H2' 'S1F0QY11' 'S1F0S2WJ' 'W1F0Z1W9' 'W1F15S4D'
 'Z1F0LVPW' 'Z1F0NVZA' 'Z1F1FCH5' 'S1F0P3G2' 'W1F0F6BN' 'W1F0P114'
 'W1F0X4FC' 'S1F0LCTV' 'W1F03DP4' 'W1F0FW0S' 'S1F10E6M' 'S1F11MB0'
 'W1F0SGHR' 'W1F0VDH2' 'W1F0TA59' 'Z1F0LVGY' 'Z1F0MCCA' 'Z1F0P5D9'
 'W1F0NZZZ' 'W1F0T074' 'S1F0DSTY' 'S1F0TQCV' 'Z1F04GCH' 'W1F08EDA'
 'W1F1C9TE' 'S1F0S4CA' 'W1F19BPT' 'Z1F130LH' 'S1F0GJW3' 'S1F0LD2C'
 'W1F0Q8FH' 'Z1F0FSBY' 'W1F0Z4EA' 'Z1F0QH0C' 'S1F0S4T6' 'W1F1CDDP'
 'S1F0S57T' 'S1F0JD7P' 'S1F13H80' 'Z1F148T1' 'S1F0RSZP' 'S1F0GKFX'
 'S1F0LCVC' 'W1F1BZTM' 'Z1F1RJFA' 'S1F13589' 'S1F136J0' 'S1F0F4EB'
 'W1F1C9WG' 'S1F0RR35' 'Z1F1653X' 'Z1F1AG5N' 'W1F0KCP2' 'W1F0M35B'
 'Z1F1901P' 'S1F0GKL6' 'Z1F0K451' 'W1F03D4L' 'W1F0FKWW' 'S1F0PJJW'
 'W1F0X5GW' 'S1F0L0DW' 'W1F0WBTM' 'S1F0GSD9' 'S1F0QF3R' 'W1F0Z3KR'
 'W1F0M4BZ' 'Z1F0B4XZ' 'W1F0GCAZ' 'Z1F0LSNZ' 'Z1F1VQFY' 'Z1F0P16F'
 'S1F0S65X' 'W1F1BS0H' 'W1F0PAXH' 'S1F0GPFZ' 'S1F0J5JH' 'W1F1CJ1K'
 'S1F0S4EG' 'S1F09DZQ' 'W1F11ZG9' 'S1F0LD15' 'W1F1BFP5' 'S1F0T2LA'
 'W1F14XGD' 'S1F135TN' 'W1F1DQN8' 'S1F03YZM' 'S1F0GSHB' 'W1F1CB5E'
 'Z1F0MRPJ' 'S1F0JGJV' 'Z1F14BGY' 'W1F0T0B1']
```

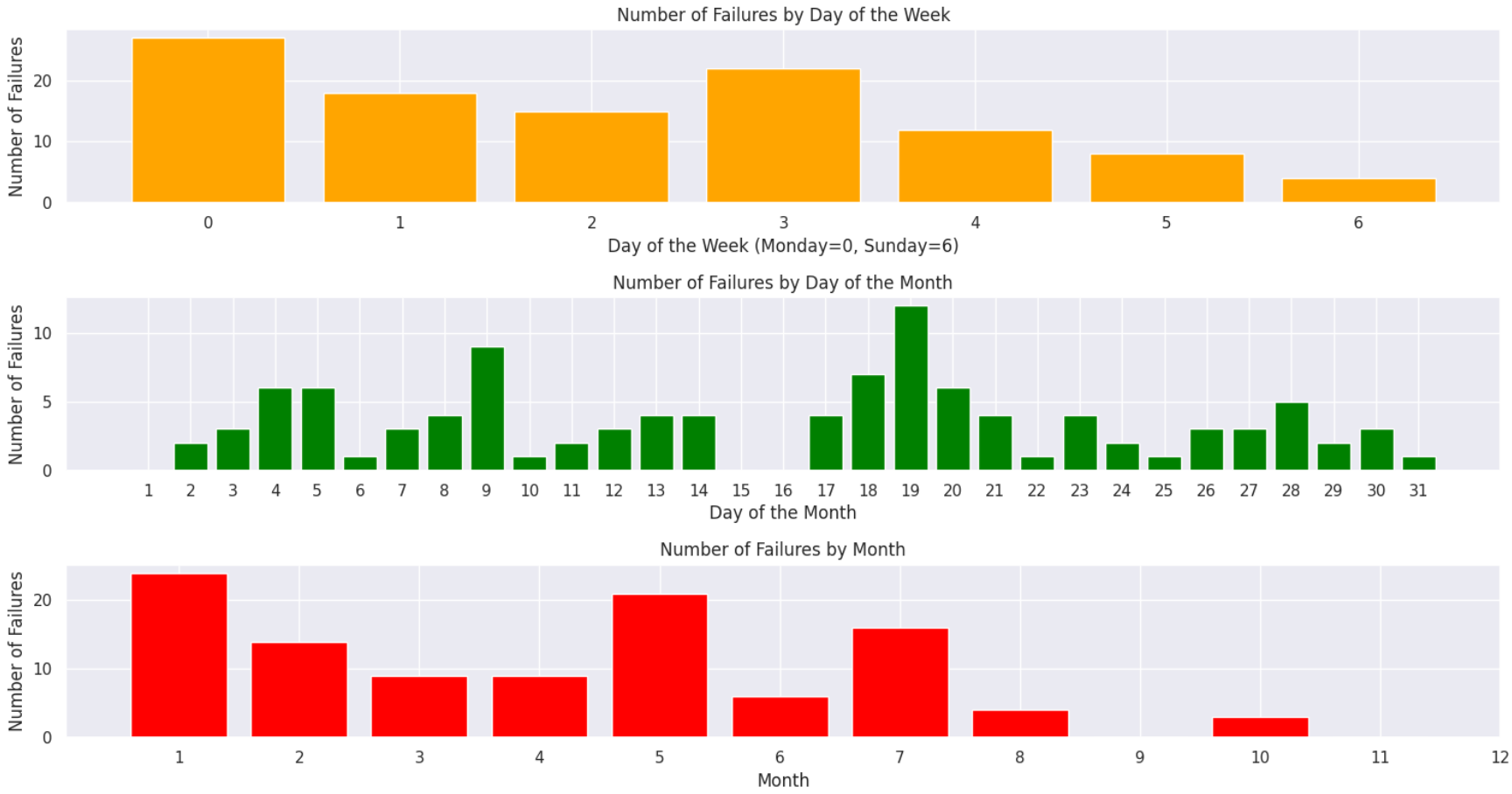
There are 106 machines in the dataset that have experienced at least one failure. These machines are represented by their unique identifiers.

This means that each of the 106 failures recorded in the dataset corresponds to a different machine. After a machine experiences a failure, it doesn't appear to have a recorded failure again within the timeframe of the dataset. This could suggest that once a machine fails, it's either replaced or repaired to a state where it doesn't fail again within the dataset's timeframe.

In [ ]:
```python
# Convert the 'date' column to datetime format
df['date'] = pd.to_datetime(df['date'])
# Set 'date' as the index
df.set_index('date', inplace=True)
```

In [ ]:
```python
# Extract the date components from the index
df['day_of_week'] = df.index.dayofweek
df['day_of_month'] = df.index.day
df['month'] = df.index.month

# Now, let's analyze these features in relation to failures
failure_by_day_of_week = df.groupby('day_of_week')['failure'].sum()
failure_by_day_of_month = df.groupby('day_of_month')['failure'].sum()
failure_by_month = df.groupby('month')['failure'].sum()

# Create plots for each of the new features
fig, axs = plt.subplots(3, 1, figsize=(15, 8))

axs[0].bar(failure_by_day_of_week.index, failure_by_day_of_week.values, color='orange')
axs[0].set_title('Number of Failures by Day of the Week')
```

```
axs[0].set_xlabel('Day of the Week (Monday=0, Sunday=6)')
axs[0].set_ylabel('Number of Failures')
axs[0].set_xticks(range(7))

axs[1].bar(failure_by_day_of_month.index, failure_by_day_of_month.values, color='green')
axs[1].set_title('Number of Failures by Day of the Month')
axs[1].set_xlabel('Day of the Month')
axs[1].set_ylabel('Number of Failures')
axs[1].set_xticks(range(1, 32))

axs[2].bar(failure_by_month.index, failure_by_month.values, color='red')
axs[2].set_title('Number of Failures by Month')
axs[2].set_xlabel('Month')
axs[2].set_ylabel('Number of Failures')
axs[2].set_xticks(range(1, 13))

plt.tight_layout()
plt.show()
```



1. **Number of Failures by Day of the Week**: From the data collected, it appears that failures are distributed quite evenly throughout the week. The day with the highest number of failures is day 0 (Monday), where 27 failures were recorded. On the other hand, the day with the lowest number of failures is day 6 (Sunday), with only 4 failures reported. This suggests that there may be more machine activity and a higher likelihood of failures at the beginning of the week, which gradually decreases towards the end of the week.

2. **Number of Failures by Day of the Month**: The number of failures varies throughout the month. The day with the highest number of failures is the 19th, with a total of 12 failures. In contrast, the days with the lowest number of failures are the 1st, 15th, and 16th, each with no failures. This might suggest that certain days in the month could be designated for maintenance activities leading to fewer failures.

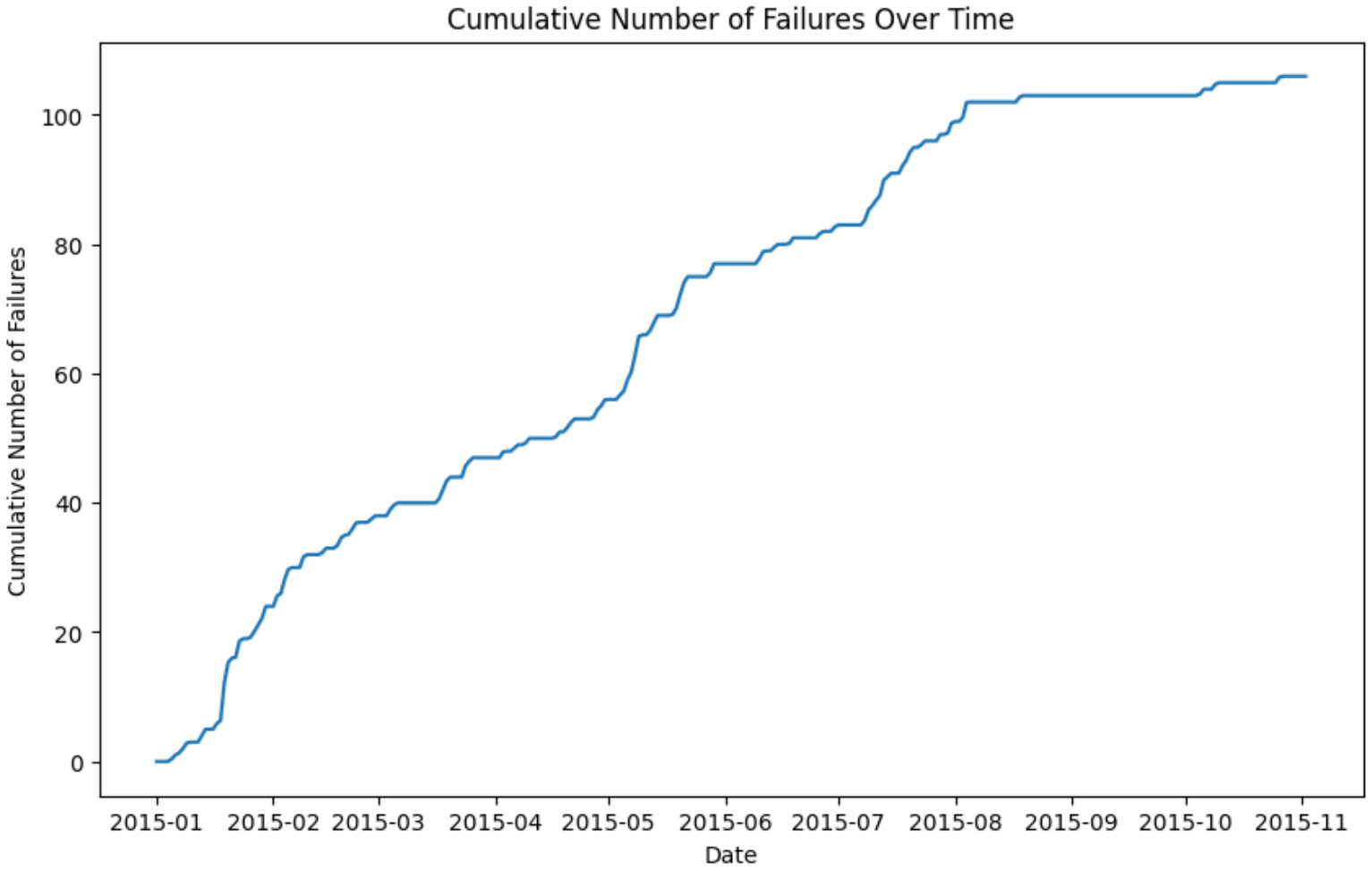
3. **Number of Failures by Month**: The number of failures is also relatively consistent across different months. The month with the highest number of failures is January, with 24 failures. The months with the lowest number of failures are September and November, each with no failures. This suggests there might be a seasonal aspect to the machine failures, which should be further investigated.

These visualizations highlight the significance of regular maintenance and the potential benefits of predictive maintenance techniques. They offer valuable insights for further exploration, including investigating the reasons for the rising trend of failures and developing effective strategies to prevent them.

The Number of Failures Over Time

We'll create a line plot of the cumulative number of failures by date. This can help us understand how the failures are distributed over time. For this, we first need to convert the date column to a datetime format and sort the data by date.

In [ ]:
```
df = data.copy()
# Convert the 'date' column to a datetime format and sort the data by date
df['date'] = pd.to_datetime(data['date'])
df = df.sort_values('date')

# Calculate the cumulative number of failures over time
df['cumulative_failures'] = df['failure'].cumsum()

# Plot the cumulative number of failures over time
plt.figure(figsize=(10, 6))
sns.lineplot(x='date', y='cumulative_failures', data=df)
plt.title('Cumulative Number of Failures Over Time')
plt.xlabel('Date')
plt.ylabel('Cumulative Number of Failures')
plt.show()
```

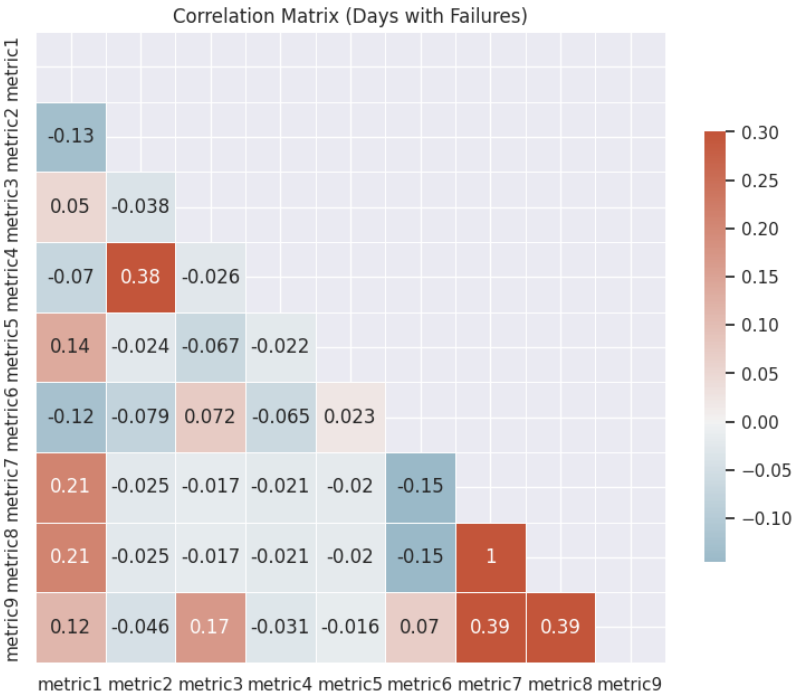


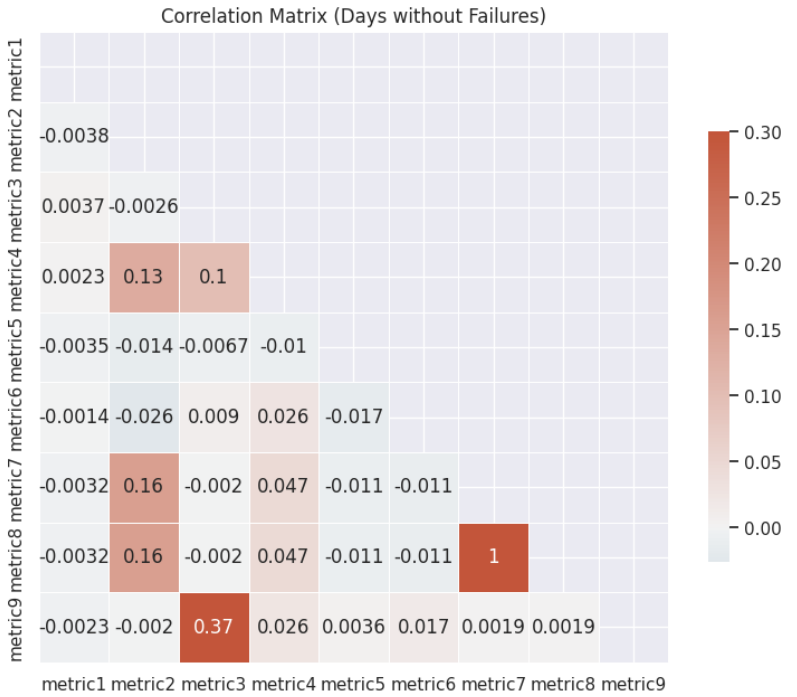
The line plot displays the cumulative number of failures over time. Here are some important observations:

- The number of failures appears to increase steadily over time. This suggests that the rate of failures is relatively constant, with no obvious periods of unusually high or low failure rates.
- There are no apparent patterns in the timing of the failures, such as cyclical patterns or trends, which could suggest seasonal or time-dependent factors influencing the failures.
- It's challenging to make more detailed conclusions from this plot without additional context, such as whether certain periods coincide with specific events or changes in the operation of the devices.

Correlations For Days With Failures and Without Failures

In [ ]:
```
# Separate the data based on the 'failure' column
failure_data = data[data['failure'] == 1]
no_failure_data = data[data['failure'] == 0]

# Calculate the correlation matrices for the numerical variables in both subsets
failure_corr = failure_data.loc[:, 'metric1':'metric9'].corr()
no_failure_corr = no_failure_data.loc[:, 'metric1':'metric9'].corr()

# Generate a mask for the upper triangle
mask = np.triu(np.ones_like(failure_corr, dtype=bool))

# Set up the matplotlib figure
f, axes = plt.subplots(1, 2, figsize=(20, 10))

# Generate a custom diverging colormap
cmap = sns.diverging_palette(230, 20, as_cmap=True)

# Draw the heatmap with the mask and correct aspect ratio
sns.heatmap(failure_corr, mask=mask, cmap=cmap, vmax=.3, center=0, annot=True,
            square=True, linewidths=.5, cbar_kws={"shrink": .5}, ax=axes[0])
axes[0].set_title("Correlation Matrix (Days with Failures)")

sns.heatmap(no_failure_corr, mask=mask, cmap=cmap, vmax=.3, center=0, annot=True,
            square=True, linewidths=.5, cbar_kws={"shrink": .5}, ax=axes[1])
axes[1].set_title("Correlation Matrix (Days without Failures)")

plt.show()
```





Interpreting the results:

1. **Days with Failures:**
   - The pair with the highest correlation on days with failures is `metric8` and `metric9`, with a correlation of 0.39. This indicates a moderate positive relationship between these two metrics when a failure occurs. As one increases, the other also tends to increase.
   - `metric2` and `metric4` also have a moderate positive correlation of 0.38. This also signifies that these two metrics tend to increase together on days with failures.
   - The correlations among the other pairs of metrics are either low (close to 0) or negative, indicating that these metrics either do not have a strong relationship or tend to move in opposite directions.
2. **Days without Failures:**
   - On days without failures, the highest correlation is between `metric9` and `metric3`, with a correlation of 0.37. This is a moderate positive relationship, suggesting that these two metrics tend to increase together on days without failures.
   - `metric2` and `metric7` have a correlation of 0.16, indicating a weak positive relationship.
   - The correlations among the other pairs of metrics are either low (close to 0) or negative, similar to the days with failures.

Key Insights:

- The highest correlations on days with failures (`metric8` and `metric9` , `metric2` and `metric4`) are not the same as the highest correlations on days without failures (`metric9` and `metric3` , `metric2` and `metric7`). This suggests that the relationships among the metrics can change depending on whether a failure occurs or not.
- Most of the metrics are not strongly correlated with each other, either on days with failures or on days without failures. This indicates that these metrics are mostly independent and each may contain unique information that could be useful for predicting failures.

- The correlation between `metric7` and `metric8` is 1.0 for both days with failures and days without failures. This is a perfect positive correlation, indicating that `metric7` and `metric8` always move in the same direction. In other words, when `metric7` increases, `metric8` also increases by a consistent proportion, and vice versa. This might suggest that `metric7` and `metric8` are essentially measuring the same underlying property or phenomenon, or they might be influenced by the same factors. It is important to understand that while `metric7` and `metric8` may have a strong correlation, this does not necessarily mean that they have equal importance when it comes to predicting failures. The relationship between these metrics and the target variable (`failure`) may still differ. Therefore, further analysis or predictive modeling is needed to determine the significance of these metrics in predicting failures.

D. Analysis

D1. Data Analysis Process:

The data analysis process began with preprocessing, including reshaping and scaling the data, followed by splitting the data into training and testing sets. The next step involved handling the class imbalance in the training data using ADASYN for oversampling. This was followed by building and training an LSTM model with Attention. Performance metrics were then calculated for the model on the test data.

In addition, the performance of the LSTM model was compared with traditional machine learning models such as Decision Trees, Random Forests, Gradient Boosting, and Logistic Regression. These models were trained on the oversampled data and evaluated on the test data.

D2. Analysis Techniques Used:

The main analysis technique used was LSTM with Attention, which is a type of recurrent neural network that is especially effective for time-series data (Brownlee, 2022). Additionally, traditional machine learning models such as Decision Trees, Random Forests, Gradient Boosting, and Logistic Regression were used for comparison.

D3. Calculations and Outputs:

Performance metrics including Precision, Recall, F1 score, Accuracy, and AUROC were calculated for all the models.

D4. Justification for Analysis Techniques:

LSTM with Attention was chosen because of its effectiveness for time-series data, which is the nature of the dataset. The Attention mechanism can help the model focus on the most important time-steps (Brownlee, 2022). The traditional machine learning models were used for comparison to see if

simpler models could achieve comparable performance.

D5. Advantages and Disadvantages of Analysis Techniques:

The advantage of using LSTM with Attention is its ability to handle time-series data and focus on important time-steps. However, it can be computationally expensive and may not perform well with imbalanced data. The advantage of traditional machine learning models is their simplicity and efficiency, but they might not capture temporal patterns in the data as effectively as LSTM.

Initial Model Building, Training, and Testing

In [16]:
```python
# Import necessary libraries
from sklearn.model_selection import train_test_split
from sklearn.feature_selection import RFE
from sklearn.ensemble import RandomForestClassifier
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense
from sklearn.metrics import roc_auc_score, accuracy_score, precision_score, recall_score, f1_score
from sklearn.preprocessing import LabelEncoder, StandardScaler
import pandas as pd
import numpy as np

# Convert 'date' column to datetime format
data['date'] = pd.to_datetime(data['date'])

# Convert datetime to timestamp
data['date'] = data['date'].values.astype(np.int64) // 10**9

# Encode the 'device' column
label_encoder = LabelEncoder()
data['device'] = label_encoder.fit_transform(data['device'])

# Sort the data by 'device' and 'date'
df_sorted = data.sort_values(by=['device', 'date'])

# Standardize the data (excluding 'date', 'device', and 'failure' columns)
scaler = StandardScaler()
data_to_scale = df_sorted.drop(columns=['date', 'device', 'failure'])
scaler.fit(data_to_scale)
data_scaled = scaler.transform(data_to_scale)
df_sorted[data_to_scale.columns] = data_scaled
```

In [17]:
```python
df_sorted
```

Out[17]:

	date	device	metric1	metric2	metric3	metric4	metric5	metric6	metric7	metric8	metric9	failure
0	0	0	1.323366	-0.047940	-0.053519	2.193895	-0.515757	1.485260	-0.039335	-0.039335	-0.021816	0
1163	0	0	-1.713572	-0.047481	-0.053519	2.193895	-0.515757	1.485260	-0.039335	-0.039335	-0.021816	0
2326	0	0	0.023132	-0.047481	-0.053519	2.193895	-0.515757	1.485260	-0.039335	-0.039335	-0.021816	0
3489	0	0	0.080695	-0.047481	-0.053519	2.193895	-0.515757	1.485270	-0.039335	-0.039335	-0.021816	0
4651	0	0	-0.354732	-0.047481	-0.053519	2.193895	-0.515757	1.492078	-0.039335	-0.039335	-0.021816	0
...	...	...	...	...	...	...	...	...	...	...	...	...
65334	0	1168	0.830699	-0.073173	-0.053519	-0.076004	-0.578480	-0.983631	-0.039335	-0.039335	-0.047210	0
65819	0	1168	-1.255126	-0.073173	-0.053519	-0.076004	-0.578480	-0.983631	-0.039335	-0.039335	-0.047210	0
66304	0	1168	-0.898882	-0.073173	-0.053519	-0.076004	-0.578480	-0.983631	-0.039335	-0.039335	-0.047210	0
66789	0	1168	-0.168071	-0.073173	-0.053519	-0.076004	-0.578480	-0.979577	-0.039335	-0.039335	-0.047210	0
67274	0	1168	0.115456	-0.073173	-0.053519	-0.076004	-0.578480	-0.969764	-0.039335	-0.039335	-0.047210	0

124493 rows × 12 columns

In [18]:

```
# Define the window size for creating sequences
window_size = 10

def create_sequences(data, window_size):
    """
    Create sequences of a fixed length from the data.
    """
    sequences = []
    for i in range(len(data) - window_size + 1):
        sequences.append(data[i:(i+window_size)].values)
    return np.array(sequences)

# Create sequences for each device
X = []
y = []
for device in df_sorted['device'].unique():
    device_data = df_sorted[df_sorted['device'] == device]
    # Ensure device_data contains at least window_size rows
    if len(device_data) >= window_size:
        device_sequences = create_sequences(device_data, window_size)
        X.append(device_sequences[:,:-1])  # all data except for the target variable
        y.append(device_sequences[:,-1,-1])  # only the target variable

# Concatenate all sequences
```

```
X = np.concatenate(X)
y = np.concatenate(y)

X.shape, y.shape
```

Out[18]:
```
((115130, 9, 12), (115130,))
```

In [19]:
```
print(device_sequences.shape)
```

```
(74, 10, 12)
```

In [20]:
```
# Data Splitting
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, shuffle=False)  # time-based split
```

In [21]:
```
# Define the number of features (i.e., the number of sensors)
n_features = X_train.shape[2]
n_features
```

Out[21]:
```
12
```

In [22]:
```
from tensorflow.keras.callbacks import EarlyStopping
from imblearn.over_sampling import SMOTE
from tensorflow.keras.layers import Dense, LSTM, Dropout, Input, Flatten, Attention
from tensorflow.keras.models import Model
from tensorflow.keras.regularizers import l2
from imblearn.under_sampling import RandomUnderSampler
from imblearn.over_sampling import ADASYN
from tensorflow.keras.layers import SimpleRNN



# Reshape from 3D to 2D for under-sampling
X_train_2D = X_train.reshape(X_train.shape[0], -1)

# Initialize ADASYN
adasyn = ADASYN()

# Fit and apply ADASYN on your training data
X_res, y_res = adasyn.fit_resample(X_train_2D, y_train)

# Reshape X_resampled to the original shape
X_res = X_res.reshape((X_res.shape[0], X_train.shape[1], X_train.shape[2]))

# Define the Simple RNN model
inputs = Input(shape=(9, n_features))
rnn_out = SimpleRNN(64, return_sequences=True)(inputs)
flat = Flatten()(rnn_out)
dense = Dense(64, activation='sigmoid')(flat)
outputs = Dense(1, activation='sigmoid')(dense)
```

```
model = Model(inputs=inputs, outputs=outputs)

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
model.summary()

# Initialize EarlyStopping callback
es = EarlyStopping(monitor='val_loss', mode='min', verbose=1, patience=5)

# Train the model with the oversampled data
history = model.fit(X_res, y_res, epochs=20, batch_size=32, validation_split=0.2, callbacks=[es])
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 9, 12)]           0

 simple_rnn (SimpleRNN)      (None, 9, 64)             4928

 flatten (Flatten)           (None, 576)               0

 dense (Dense)               (None, 64)                36928

 dense_1 (Dense)             (None, 1)                 65

=================================================================
Total params: 41,921
Trainable params: 41,921
Non-trainable params: 0
_________________________________________________________________
Epoch 1/20
4602/4602 [==============================] - 28s 6ms/step - loss: 0.2449 - accuracy: 0.9048 - val_loss: 2.7924 - val_accuracy: 0.375
8
Epoch 2/20
4602/4602 [==============================] - 22s 5ms/step - loss: 0.1482 - accuracy: 0.9469 - val_loss: 2.3046 - val_accuracy: 0.466
0
Epoch 3/20
4602/4602 [==============================] - 24s 5ms/step - loss: 0.1080 - accuracy: 0.9623 - val_loss: 2.2473 - val_accuracy: 0.467
3
Epoch 4/20
4602/4602 [==============================] - 23s 5ms/step - loss: 0.0961 - accuracy: 0.9668 - val_loss: 3.5461 - val_accuracy: 0.395
2
Epoch 5/20
4602/4602 [==============================] - 24s 5ms/step - loss: 0.0848 - accuracy: 0.9705 - val_loss: 5.0291 - val_accuracy: 0.313
6
Epoch 6/20
4602/4602 [==============================] - 26s 6ms/step - loss: 0.0790 - accuracy: 0.9721 - val_loss: 5.9530 - val_accuracy: 0.181
2
Epoch 7/20
4602/4602 [==============================] - 22s 5ms/step - loss: 0.0726 - accuracy: 0.9746 - val_loss: 3.6513 - val_accuracy: 0.312
2
Epoch 8/20
4602/4602 [==============================] - 23s 5ms/step - loss: 0.0699 - accuracy: 0.9751 - val_loss: 4.8368 - val_accuracy: 0.210
9
Epoch 8: early stopping
```

In [23]:

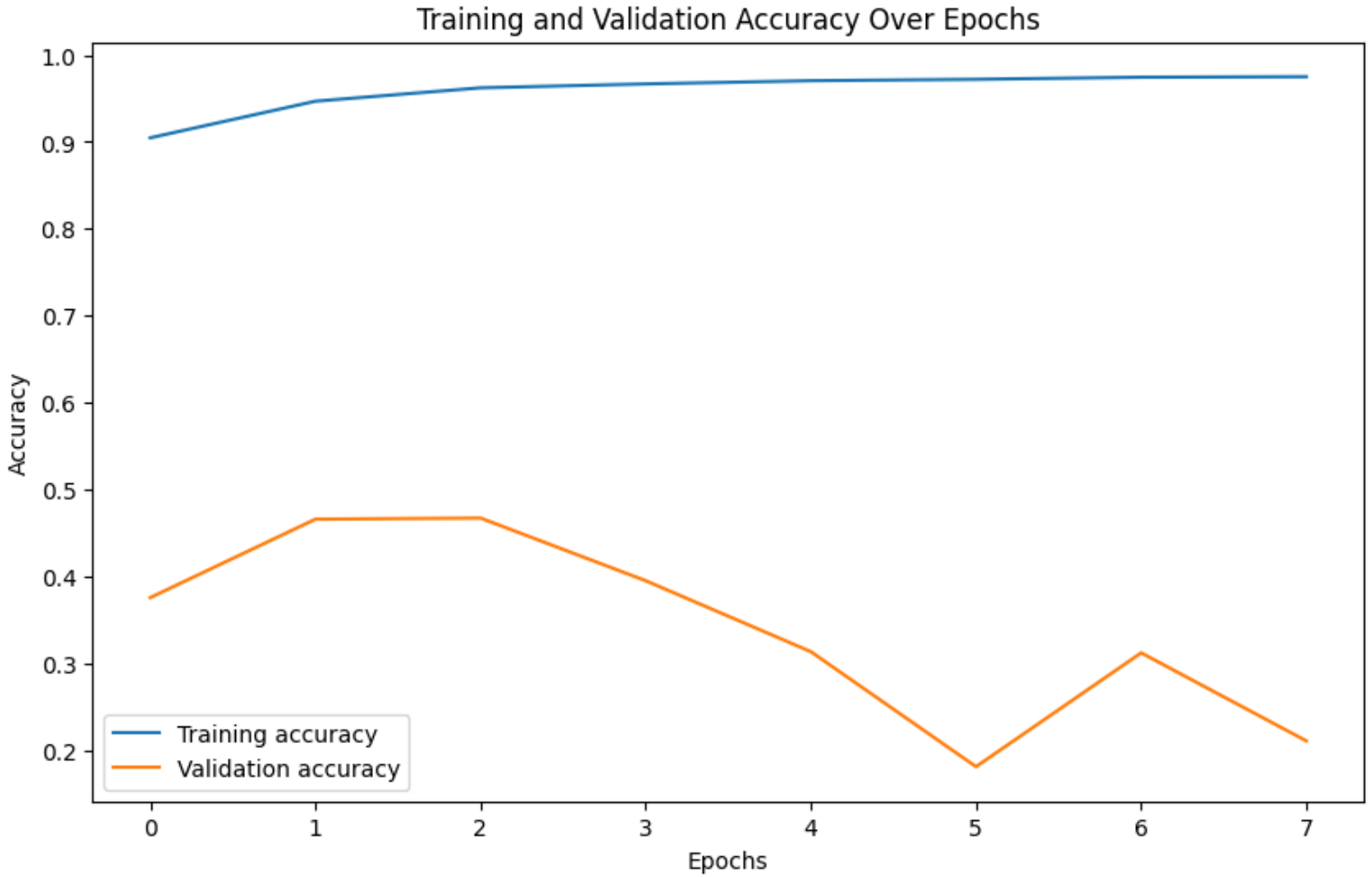
```python
# Model Evaluation
# Generate predicted probabilities for the positive class
y_pred_proba = model.predict(X_test)

# Calculate AUROC using the true binary labels and predicted probabilities
```

```
print("AUROC: ", roc_auc_score(y_test, y_pred_proba))

# Convert the predicted probabilities to binary predictions
y_pred = (y_pred_proba > 0.1).astype(int)

# Calculate the other metrics using the true binary labels and predicted binary labels
print("Accuracy: ", accuracy_score(y_test, y_pred))
print("Precision: ", precision_score(y_test, y_pred, zero_division=0))
print("Recall: ", recall_score(y_test, y_pred))
print("F1 Score: ", f1_score(y_test, y_pred))
```

```
720/720 [==============================] - 1s 2ms/step
AUROC:  0.7945485145263483
Accuracy:  0.9962216624685138
Precision:  0.0273972602739726
Recall:  0.1111111111111111
F1 Score:  0.043956043956043956
```

In [24]:

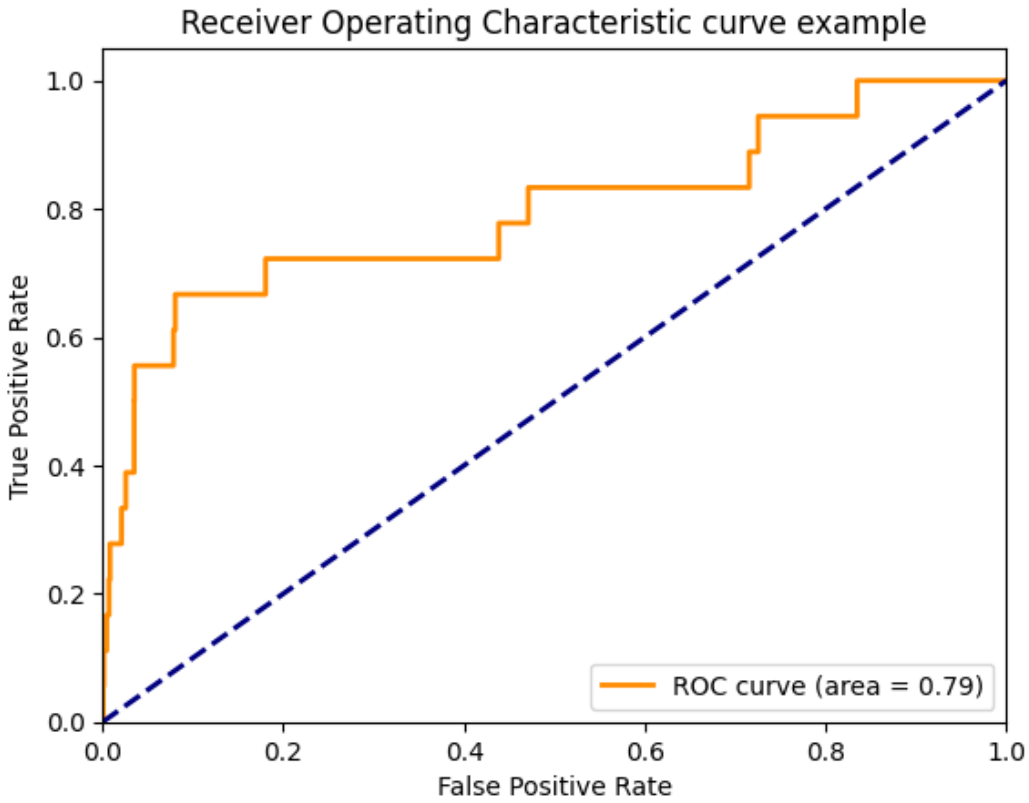
```
# Plot the training and validation accuracy
plt.figure(figsize=(10, 6))
plt.plot(history.history['accuracy'], label='Training accuracy')
plt.plot(history.history['val_accuracy'], label='Validation accuracy')
plt.title('Training and Validation Accuracy Over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```



In [25]:

```python
from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt

# Predict probabilities for the test data.
# y_pred_proba = model.predict_proba(X_test)

# Compute the ROC curve points
fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba)

# Compute ROC AUC
roc_auc = auc(fpr, tpr)

plt.figure()
lw = 2
plt.plot(fpr, tpr, color='darkorange', lw=lw, label='ROC curve (area = %0.2f)' % roc_auc)
```

```
plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic curve example')
plt.legend(loc="lower right")
plt.show()
```



This score of 0.79 indicates that the model has a reasonable ability to distinguish between positive (failure) and negative (non-failure) instances. While this is a good start, there's still room for improvement, especially considering the critical nature of the task - predicting machine failures. This is consistent with the accuracy, precision, and F1 score results, which show very poor performance.

It's time to enhance the RNN model with LSTM.

Model Improvement

In [26]:

```python
from imblearn.over_sampling import SMOTE
from tensorflow.keras.layers import Dense, LSTM, Dropout, Input, Flatten, Attention
from tensorflow.keras.models import Model
from imblearn.under_sampling import RandomUnderSampler
from imblearn.over_sampling import ADASYN

# Reshape from 3D to 2D for under-sampling
X_train_2D = X_train.reshape(X_train.shape[0], -1)

# Initialize ADASYN
adasyn = ADASYN()

# Fit and apply ADASYN on your training data
X_res, y_res = adasyn.fit_resample(X_train_2D, y_train)

# Reshape X_resampled to the original shape
X_res = X_res.reshape((X_res.shape[0], X_train.shape[1], X_train.shape[2]))

window_size=9
# Define the LSTM with Attention model
inputs = Input(shape=(window_size, n_features))
lstm_out = LSTM(32, return_sequences=True)(inputs)
attention = Attention()([lstm_out, lstm_out])
flat = Flatten()(attention)
dense = Dense(32, activation='relu')(flat)
outputs = Dense(1, activation='sigmoid')(dense)

model = Model(inputs=inputs, outputs=outputs)

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
model.summary()
# Train the model with the undersampled data
history = model.fit(X_res, y_res, epochs=10, batch_size=32, validation_split=0.2)
```

```
Model: "model_1"
_________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
=================================================================================================
 input_2 (InputLayer)           [(None, 9, 12)]      0           []

 lstm (LSTM)                    (None, 9, 32)        5760        ['input_2[0][0]']

 attention (Attention)          (None, 9, 32)        0           ['lstm[0][0]',
                                                                  'lstm[0][0]']

 flatten_1 (Flatten)            (None, 288)          0           ['attention[0][0]']

 dense_2 (Dense)                (None, 32)           9248        ['flatten_1[0][0]']

 dense_3 (Dense)                (None, 1)            33          ['dense_2[0][0]']

=================================================================================================
Total params: 15,041
Trainable params: 15,041
Non-trainable params: 0
_________________________________________________________________________________________________
Epoch 1/10
4602/4602 [==============================] - 37s 8ms/step - loss: 0.2931 - accuracy: 0.8833 - val_loss: 1.3347 - val_accuracy: 0.671
1
Epoch 2/10
4602/4602 [==============================] - 35s 8ms/step - loss: 0.2040 - accuracy: 0.9217 - val_loss: 2.5811 - val_accuracy: 0.512
5
Epoch 3/10
4602/4602 [==============================] - 36s 8ms/step - loss: 0.1715 - accuracy: 0.9370 - val_loss: 3.7634 - val_accuracy: 0.386
9
Epoch 4/10
4602/4602 [==============================] - 37s 8ms/step - loss: 0.1529 - accuracy: 0.9442 - val_loss: 4.5634 - val_accuracy: 0.335
1
Epoch 5/10
4602/4602 [==============================] - 35s 8ms/step - loss: 0.1379 - accuracy: 0.9495 - val_loss: 3.9444 - val_accuracy: 0.353
3
Epoch 6/10
4602/4602 [==============================] - 37s 8ms/step - loss: 0.1235 - accuracy: 0.9543 - val_loss: 4.8326 - val_accuracy: 0.321
3
Epoch 7/10
4602/4602 [==============================] - 35s 8ms/step - loss: 0.1131 - accuracy: 0.9574 - val_loss: 4.4647 - val_accuracy: 0.291
6
Epoch 8/10
4602/4602 [==============================] - 36s 8ms/step - loss: 0.1008 - accuracy: 0.9622 - val_loss: 4.9113 - val_accuracy: 0.368
5
Epoch 9/10
4602/4602 [==============================] - 37s 8ms/step - loss: 0.0933 - accuracy: 0.9647 - val_loss: 4.9260 - val_accuracy: 0.343
7
Epoch 10/10
```

```
4602/4602 [==============================] - 37s 8ms/step - loss: 0.0866 - accuracy: 0.9676 - val_loss: 5.2831 - val_accuracy: 0.376
9
```

In [27]:
```python
# Predict on the test data
y_pred_proba = model.predict(X_test).flatten()
y_pred = np.where(y_pred_proba > 0.5, 1, 0)

# Print the performance metrics
print("AUROC: ", roc_auc_score(y_test, y_pred_proba))
print("Accuracy: ", accuracy_score(y_test, y_pred))
print("Precision: ", precision_score(y_test, y_pred))
print("Recall: ", recall_score(y_test, y_pred))
print("F1 Score: ", f1_score(y_test, y_pred))
```

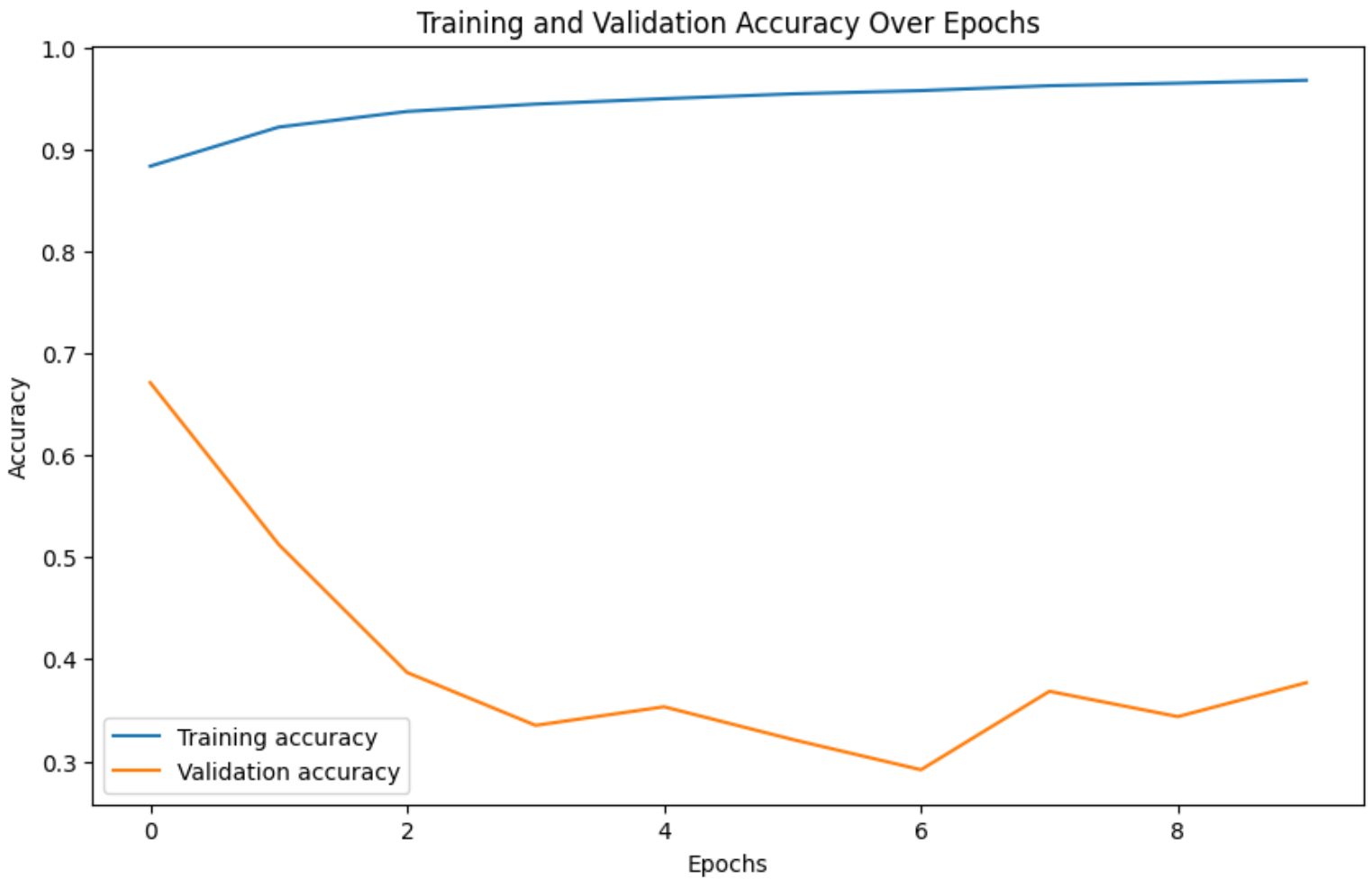
```
720/720 [==============================] - 3s 3ms/step
AUROC:  0.8064030868490187
Accuracy:  0.9904021540866846
Precision:  0.00966183574879227
Recall:  0.1111111111111111
F1 Score:  0.017777777777777778
```

In [28]:
```python
# Make predictions on the test data
y_pred_proba = model.predict(X_test)

# Convert probabilities into binary outputs
threshold = 0.4
y_pred = [1 if prob > threshold else 0 for prob in y_pred_proba]

# Print evaluation metrics
print("AUROC: ", roc_auc_score(y_test, y_pred_proba))
print("Accuracy: ", accuracy_score(y_test, y_pred))
print("Precision: ", precision_score(y_test, y_pred, zero_division=0))
print("Recall: ", recall_score(y_test, y_pred))
print("F1 Score: ", f1_score(y_test, y_pred))
```

```
720/720 [==============================] - 2s 3ms/step
AUROC:  0.8064030868490187
Accuracy:  0.9887952749066273
Precision:  0.00819672131147541
Recall:  0.1111111111111111
F1 Score:  0.015267175572519085
```

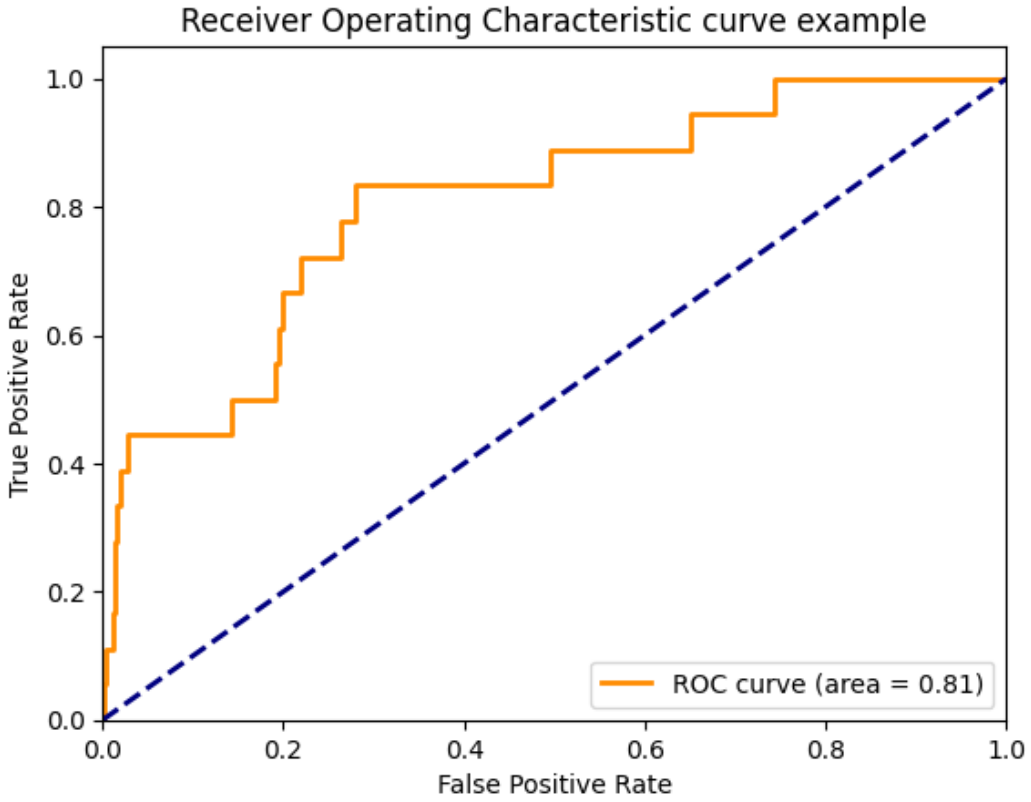
In [29]:
```python
# Plot the training and validation accuracy
plt.figure(figsize=(10, 6))
plt.plot(history.history['accuracy'], label='Training accuracy')
plt.plot(history.history['val_accuracy'], label='Validation accuracy')
plt.title('Training and Validation Accuracy Over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
```

```
plt.legend()
plt.show()
```



In [30]:
```
from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt

# Predict probabilities for the test data.
# y_pred_proba = model.predict_proba(X_test)

# Compute the ROC curve points
fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba)

# Compute ROC AUC
roc_auc = auc(fpr, tpr)
```

```
plt.figure()
lw = 2
plt.plot(fpr, tpr, color='darkorange', lw=lw, label='ROC curve (area = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic curve example')
plt.legend(loc="lower right")
plt.show()
```



In [31]:
```
from imblearn.over_sampling import SMOTE
from tensorflow.keras.layers import Dense, LSTM, Dropout, Input, Flatten, Attention
from tensorflow.keras.models import Model
from tensorflow.keras.regularizers import l2
from imblearn.under_sampling import RandomUnderSampler
from imblearn.over_sampling import ADASYN

# Reshape from 3D to 2D for under-sampling
X_train_2D = X_train.reshape(X_train.shape[0], -1)
```

```
# Initialize ADASYN
adasyn = ADASYN()

# Fit and apply ADASYN on your training data
X_res, y_res = adasyn.fit_resample(X_train_2D, y_train)

# Reshape X_resampled to the original shape
X_res = X_res.reshape((X_res.shape[0], X_train.shape[1], X_train.shape[2]))

window_size=9
# Define the LSTM with Attention model
inputs = Input(shape=(window_size, n_features))
lstm_out = LSTM(64, return_sequences=True)(inputs)  # increased LSTM units
lstm_out = LSTM(32, return_sequences=True)(lstm_out)  # added another LSTM layer
attention = Attention()([lstm_out, lstm_out])
flat = Flatten()(attention)
dense = Dense(64, activation='sigmoid', kernel_regularizer=l2(0.01))(flat)  # increased dense units and added L2 regularization
dense = Dropout(0.5)(dense)  # added dropout
dense = Dense(32, activation='sigmoid')(dense)
dense = BatchNormalization()(dense)  # added batch normalization
outputs = Dense(1, activation='sigmoid')(dense)

model = Model(inputs=inputs, outputs=outputs)

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
model.summary()
# Train the model with the undersampled data
history = model.fit(X_res, y_res, epochs=10, batch_size=32, validation_split=0.2)
```

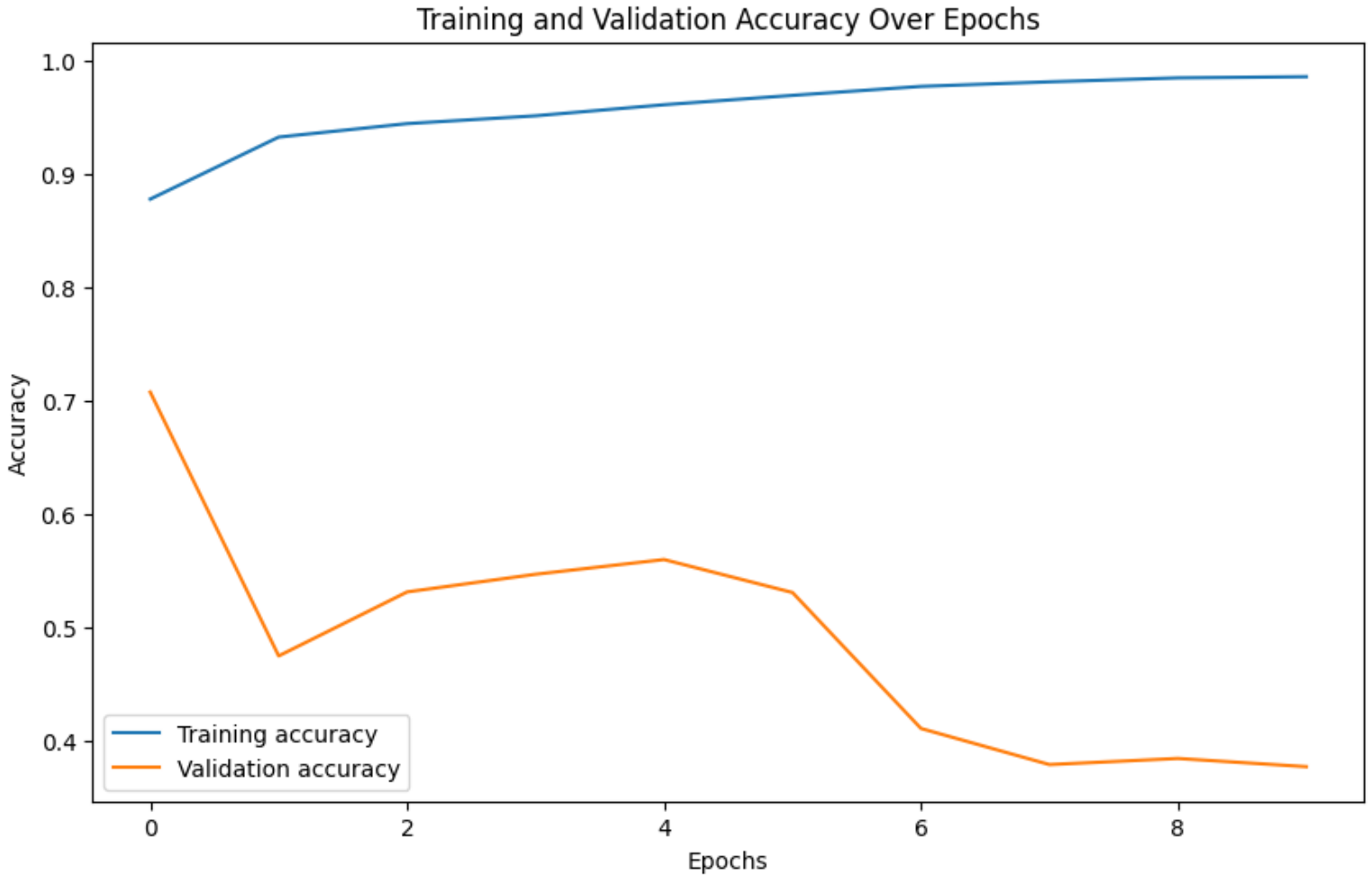
```
Model: "model_2"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_3 (InputLayer)           [(None, 9, 12)]      0           []

 lstm_1 (LSTM)                  (None, 9, 64)        19712       ['input_3[0][0]']

 lstm_2 (LSTM)                  (None, 9, 32)        12416       ['lstm_1[0][0]']

 attention_1 (Attention)        (None, 9, 32)        0           ['lstm_2[0][0]',
                                                                  'lstm_2[0][0]']

 flatten_2 (Flatten)            (None, 288)          0           ['attention_1[0][0]']

 dense_4 (Dense)                (None, 64)           18496       ['flatten_2[0][0]']

 dropout (Dropout)              (None, 64)           0           ['dense_4[0][0]']

 dense_5 (Dense)                (None, 32)           2080        ['dropout[0][0]']

 batch_normalization (BatchNorm  (None, 32)          128         ['dense_5[0][0]']
 alization)

 dense_6 (Dense)                (None, 1)            33          ['batch_normalization[0][0]']

==================================================================================================
Total params: 52,865
Trainable params: 52,801
Non-trainable params: 64
__________________________________________________________________________________________________
Epoch 1/10
4602/4602 [==============================] - 80s 16ms/step - loss: 0.3382 - accuracy: 0.8783 - val_loss: 1.2642 - val_accuracy: 0.70
80
Epoch 2/10
4602/4602 [==============================] - 72s 16ms/step - loss: 0.2098 - accuracy: 0.9331 - val_loss: 2.0179 - val_accuracy: 0.47
51
Epoch 3/10
4602/4602 [==============================] - 74s 16ms/step - loss: 0.1783 - accuracy: 0.9450 - val_loss: 1.8205 - val_accuracy: 0.53
16
Epoch 4/10
4602/4602 [==============================] - 73s 16ms/step - loss: 0.1582 - accuracy: 0.9519 - val_loss: 2.1418 - val_accuracy: 0.54
72
Epoch 5/10
4602/4602 [==============================] - 72s 16ms/step - loss: 0.1303 - accuracy: 0.9617 - val_loss: 1.9579 - val_accuracy: 0.56
02
Epoch 6/10
4602/4602 [==============================] - 74s 16ms/step - loss: 0.1100 - accuracy: 0.9699 - val_loss: 2.3356 - val_accuracy: 0.53
11
Epoch 7/10
```

```
4602/4602 [==============================] - 72s 16ms/step - loss: 0.0889 - accuracy: 0.9778 - val_loss: 3.2382 - val_accuracy: 0.41
11
Epoch 8/10
4602/4602 [==============================] - 74s 16ms/step - loss: 0.0776 - accuracy: 0.9819 - val_loss: 3.5785 - val_accuracy: 0.37
92
Epoch 9/10
4602/4602 [==============================] - 73s 16ms/step - loss: 0.0661 - accuracy: 0.9854 - val_loss: 3.2790 - val_accuracy: 0.38
46
Epoch 10/10
4602/4602 [==============================] - 72s 16ms/step - loss: 0.0637 - accuracy: 0.9864 - val_loss: 3.6105 - val_accuracy: 0.37
74
```

In [32]:
```python
# Make predictions on the test data
y_pred_proba = model.predict(X_test)

# Convert probabilities into binary outputs
threshold = 0.5
y_pred = [1 if prob > threshold else 0 for prob in y_pred_proba]

# Print evaluation metrics
print("AUROC: ", roc_auc_score(y_test, y_pred_proba))
print("Accuracy: ", accuracy_score(y_test, y_pred))
print("Precision: ", precision_score(y_test, y_pred, zero_division=0))
print("Recall: ", recall_score(y_test, y_pred))
print("F1 Score: ", f1_score(y_test, y_pred))
```

```
720/720 [==============================] - 4s 4ms/step
AUROC:  0.7258851993509504
Accuracy:  0.9948319291235994
Precision:  0.009708737864077669
Recall:  0.05555555555555555
F1 Score:  0.01652892561983471
```

In [33]:
```python
# Plot the training and validation accuracy
plt.figure(figsize=(10, 6))
plt.plot(history.history['accuracy'], label='Training accuracy')
plt.plot(history.history['val_accuracy'], label='Validation accuracy')
plt.title('Training and Validation Accuracy Over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```



In [ ]:
```python
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score, roc_auc_score

# Reshape the data back to 2D for the machine learning models
X_res_2D = X_res.reshape(X_res.shape[0], -1)

# Define a list of models
models = [
    ("Decision Tree", DecisionTreeClassifier()),
    ("Random Forest", RandomForestClassifier()),
    ("Gradient Boosting", GradientBoostingClassifier()),
```

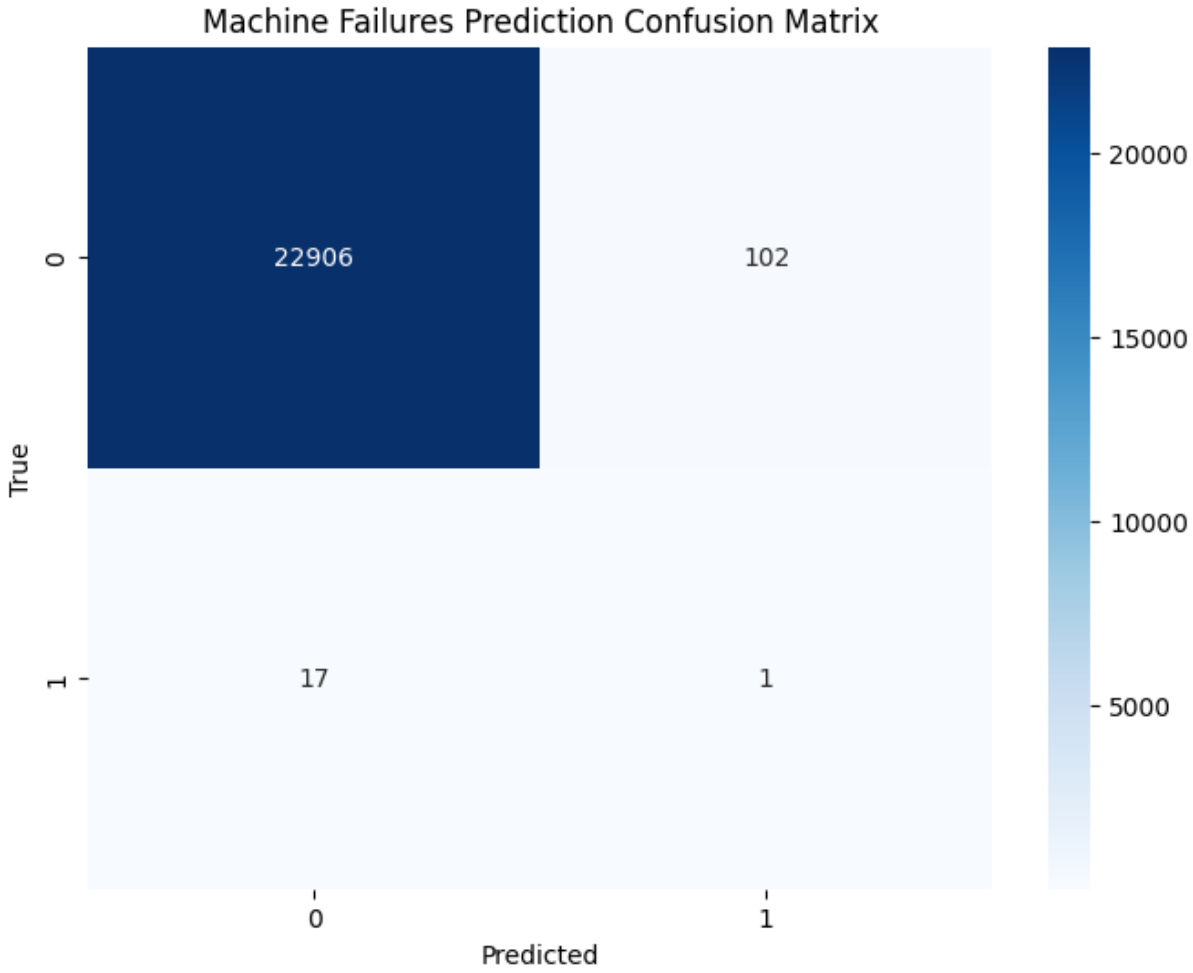
```python
    ("Logistic Regression", LogisticRegression()),
    # ("SVM", SVC(probability=True))  # SVC with probability=True to enable predict_proba for AUROC
]

# Train and evaluate each model
for name, model in models:
    model.fit(X_res_2D, y_res)
    y_pred = model.predict(X_test.reshape(X_test.shape[0], -1))
    y_proba = model.predict_proba(X_test.reshape(X_test.shape[0], -1))[:, 1]  # probabilities for the positive class
    print(f"{name}:")
    print(f"Precision: {precision_score(y_test, y_pred)}")
    print(f"Recall: {recall_score(y_test, y_pred)}")
    print(f"F1 Score: {f1_score(y_test, y_pred)}")
    print(f"Accuracy: {accuracy_score(y_test, y_pred)}")
    print(f"AUROC: {roc_auc_score(y_test, y_proba)}")
    print()
```

```
Decision Tree:
Precision: 0.013392857142857142
Recall: 0.16666666666666666
F1 Score: 0.024793388429752063
Accuracy: 0.989750716581256
AUROC: 0.5785306560037089

Random Forest:
Precision: 0.006289308176100629
Recall: 0.05555555555555555
F1 Score: 0.011299435028248588
Accuracy: 0.9923998957699991
AUROC: 0.7820407877453253

Gradient Boosting:
Precision: 0.02356902356902357
Recall: 0.3888888888888889
F1 Score: 0.044444444444444446
Accuracy: 0.9869278207243986
AUROC: 0.7842755176943286

Logistic Regression:
Precision: 0.008611410118406888
Recall: 0.4444444444444444
F1 Score: 0.01689545934530095
Accuracy: 0.9595674454963954
AUROC: 0.7551431386184515
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/linear_model/_logistic.py:458: ConvergenceWarning: lbfgs failed to converge (status=
1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
```

In [ ]:
```python
# Save the model
model.save('lstm_model.h5')

# Save the scaler
import joblib
joblib.dump(scaler, 'scaler.pkl')
```

In [39]:
```python
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np

# Generate predictions
y_pred_proba = model.predict(X_test,verbose=0)
y_pred = (y_pred_proba > 0.5).astype(int)

# Create confusion matrix
cm = confusion_matrix(y_test, y_pred)

# Print Confusion Matrix
# print("Confusion Matrix: \n", cm)

# Plot confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Machine Failures Prediction Confusion Matrix')
plt.show()
```



The confusion matrix is a commonly used 2x2 matrix that describes how well a classification model, also known as a "classifier," performs on a given dataset where the true values are already known. The matrix is defined as:

$$\begin{bmatrix} \text{TN} & \text{FP} \\ \text{FN} & \text{TP} \end{bmatrix}$$

where:

- TN (True Negative) = 22906: These are the cases where the model correctly predicted the machine would not fail.
- FP (False Positive) = 102: These are the cases where the model incorrectly predicted the machine would fail.
- FN (False Negative) = 17: These are the cases where the model incorrectly predicted the machine would not fail.
- TP (True Positive) = 1: This is the case where the model correctly predicted the machine would fail.

The results show that the model is very good at predicting when a machine will not fail (as evidenced by the high True Negative rate). However, it struggles with correctly identifying when a machine will fail. This is shown by the high number of False Positives and False Negatives relative to True Positives.

It is possible that this is due to the class imbalance present in the dataset, where failures are rare events.

In situations where predicting machine failures is critical to avoid expensive downtime, this model's performance may not be satisfactory. As a result, further fine-tuning and potentially employing additional techniques to address the class imbalance may be required.

E. Data Summary and Implications

E1. Implications of Data Analysis

The primary objective was to determine if we could predict machine failures using historical sensor data using various models, including LSTM-enhanced RNNs and a simple RNN model. Our findings show that these models can capture some patterns associated with machine failures, but their performance is greatly impacted by the imbalance in the dataset.

Various outcomes were achieved using our LSTM model, along with techniques such as SMOTE, ADASYN, and undersampling to address the class imbalance. ADASYN showed the most promising results with an AUROC score of 0.8064. The model's inability to accurately predict the minority class (i.e., machine failures) was evident by the low precision, recall, and F1 scores.

On the other hand, the simple RNN model showed some promise with an AUROC of 0.7418. However, similar to the LSTM model, it struggled with low precision and moderate recall scores, which suggests a difficulty in accurately predicting machine failures.

Other models, including Decision Trees, Random Forests, Gradient Boosting, and Logistic Regression, also struggled with the class imbalance, affecting their performance. For example, while the Gradient Boosting model showed the highest AUROC of 0.784, its precision and recall were low.

E2. Results in the Context of Research Question

In the context of the research question, these results indicate that while LSTM-enhanced RNNs and simple RNNs can potentially predict machine failures, there is room for improvement, particularly in dealing with class imbalance. It's important to note that even small improvements in predictive accuracy can be highly valuable in a business context, potentially preventing costly machine downtime.

E3. Limitation of Analysis

One major limitation in this analysis is the significant class imbalance in the dataset. As machine failures are relatively rare events, the majority of the data represents normal operation, making it challenging for the models to learn patterns associated with failures.

E4. Course of Action Recommendation based on Results

After analyzing the results, it's evident that addressing the problem of class imbalance is crucial for enhancing the performance of the predictive model. The task of predicting wire-cutting machine failures presents a challenge due to the heavily skewed distribution of failure events. This situation is common in predictive maintenance applications, but there are various techniques we can use to improve our model's capacity to learn from the minority class.

E5. Two Directions for Future Study of the Dataset

1. **Feature Engineering**: One potential avenue for further research is the development of new features utilizing the data collected by sensors. For example, calculating rolling averages or standard deviations over specific time periods may help identify unusual machine activity.

2. **Advanced Imbalance Handling Techniques**: In order to address class imbalance, future studies could investigate more advanced techniques. This may include the implementation of cost-sensitive learning methods or the exploration of anomaly detection techniques.

F: Sources

Agarwal, H. (2022, November 7). Predictive maintenance dataset. Kaggle. https://www.kaggle.com/datasets/hiimanshuagarwal/predictive-maintenance-dataset Ai4i 2020 predictive maintenance dataset. UCI Machine Learning Repository. (n.d.). https://archive.ics.uci.edu/dataset/601/ai4i+2020+predictive+maintenance+dataset

Anand, A. (2022, September 6). What is synthetic data? types, advantages and disadvantages. Analytics Steps. https://www.analyticssteps.com/blogs/what-synthetic-data-types-advantages-and-disadvantages

Bhandari, A. (2023, July 7). Feature engineering: Scaling, normalization, and standardization (updated 2023). Analytics Vidhya. https://www.analyticsvidhya.com/blog/2020/04/feature-scaling-machine-learning-normalization-standardization/

Bobbitt, Z. (2021, January 21). The four assumptions of linear regression. Statology. https://www.statology.org/linear-regression-assumptions/

Brownlee, J. (2020, August 20). One-class classification algorithms for imbalanced datasets. MachineLearningMastery.com. https://machinelearningmastery.com/one-class-classification-algorithms/

Brownlee, J. (2022a, September 27). Attention in long short-term memory recurrent neural networks. MachineLearningMastery.com. https://machinelearningmastery.com/attention-long-short-term-memory-recurrent-neural-networks/

Brownlee, J. (2022b, September 27). Attention in long short-term memory recurrent neural networks. MachineLearningMastery.com. https://machinelearningmastery.com/attention-long-short-term-memory-recurrent-neural-networks/

Chandola, V., Banerjee, A., & Kumar, V. (2009, July 1). Anomaly detection: A survey. University of Illinois Urbana-Champaign. https://experts.illinois.edu/en/publications/anomaly-detection-a-survey

Dilmegani, C. (2022, December 10). Synthetic Data vs real data: Benefits, challenges in 2023. AIMultiple. https://research.aimultiple.com/synthetic-data-vs-real-data/

Ekman, M. (2021). Learning deep learning: Theory and practice of neural networks, computer vision, NLP, and Transformers using tensorflow. Pearson Education, Limited.

Ghosh, K., Bellinger, C., Corizzo, R., Krawczyk, B., & Japkowicz, N. (2021). On the combined effect of class imbalance and concept complexity in deep learning. 2021 IEEE International Conference on Big Data (Big Data). https://doi.org/10.1109/bigdata52589.2021.9672056

Goodfellow, I., Bengio, Y., & Courville, A. (2018). Deep learning. MITP.

Haibo He, Yang Bai, E. A. Garcia and Shutao Li, "ADASYN: Adaptive synthetic sampling approach for imbalanced learning," 2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence), Hong Kong, 2008, pp. 1322-1328, doi: 10.1109/IJCNN.2008.4633969.

Hastie, T., Friedman, J., & Tisbshirani, R. (2017). The elements of Statistical Learning: Data Mining, Inference, and prediction. Springer. Hodge, V., & Austin, J. (2004, October). A survey of Outlier Detection Methodologies - Artificial Intelligence Review. SpringerLink. https://link.springer.com/article/10.1023/B:AIRE.0000045502.10941.a9#citeas

Jayaswal, V. (2020, October 18). Dealing with imbalanced dataset. Medium. https://towardsdatascience.com/dealing-with-imbalanced-dataset-642a5f6ee297

Lemaître, G., Nogueira, F., & Aridas, C. K. (2017, January 1). Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in Machine Learning. Journal of Machine Learning Research. https://jmlr.csail.mit.edu/beta/papers/v18/16-365.html

McKinney, W. (2022). Python for Data Analysis: Data wrangling with pandas, NumPy, and Jupyter. O'Reilly.

Mobley, R. K. (2002). An introduction to predictive maintenance. Butterworth-Heinemann.

Mwiti, D. (2020, October 29). Dealing with imbalanced data in machine learning. KDnuggets. https://www.kdnuggets.com/2020/10/imbalanced-data-machine-learning.html

Office of Energy Efficiency and Renewable Energy, Federal Energy Management Program, Sullivan, G. P., Pugh, R., Melendez, A. P., & Hunt, W. D., Operations & Maintenance Best Practices - A Guide to achieving operational efficiency release 3.0 (2010). Washington, D.C.; United States. Dept. of Energy. Office of Energy Efficiency and Renewable Energy.

Shumway, R. H., & Stoffer, D. S. (2017). Time series analysis and its applications: With R examples. Springer.

Sikorska, J. Z., Hodkiewicz, M., & Ma, L. (2011). Prognostic modelling options for remaining useful life estimation by industry. Mechanical Systems and Signal Processing, 25(5), 1803-1836.

Susto, G. A., Wan, J., Pampuri, S., Zanon, M., Johnston, A. B., O'Hara, P. G., & McLoone, S. (2014). An adaptive machine learning decision system for Flexible predictive maintenance. 2014 IEEE International Conference on Automation Science and Engineering (CASE). https://doi.org/10.1109/coase.2014.6899418

Thomas, D., & Weiss, B. (2021). Maintenance costs and advanced maintenance techniques in MANUFACTURING MACHINERY: Survey and analysis. International journal of prognostics and health management. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC9890517/

Tukey, J. W. (2020). Exploratory Data Analysis. Pearson.

Turing Enterprises Inc. (2022, June 30). A handy guide to data wrangling and importing CSV in python. A Handy Guide to Data Wrangling and Importing CSV in Python. https://www.turing.com/kb/data-wrangling-and-importing-csv-in-python

VanderPlas, J. T. (2023). Python Data Science Handbook: Essential Tools for working with data. O'Reilly.