Zephyr: a Data-Centric Framework for Predictive Maintenance of Wind Turbines

by

Frances R. Hartwell

S.B. Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 2019

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2023

© Massachusetts Institute of Technology 2023. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
January 20, 2023

Certified by . . . .
Kalyan Veeramachaneni
Principal Research Scientist
Thesis Supervisor

Accepted by . . . .
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Zephyr: a Data-Centric Framework for Predictive Maintenance of Wind Turbines

by

Frances R. Hartwell

Submitted to the Department of Electrical Engineering and Computer Science
on January 20, 2023, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Because wind turbines often operate through harsh weather events, under variable operating conditions, and in difficult-to-access locations, turbine maintenance is often challenging and costly. In this thesis, we present Zephyr, a flexible machine learning framework for predictive maintenance of wind energy assets. Manual analysis of wind turbine data is difficult and time-consuming due to its volume, variety, and, most importantly, the need for quick detection of issues. Machine learning (ML) methods are able to automate large-scale data analysis. However, the enormous amount of contextual information required to actually understand the data impedes the ability of ML frameworks to provide actionable insights. To this end, Zephyr enables Subject Matter Experts (SMEs) to incorporate their knowledge at various stages of ML model development. The Zephyr framework consists of a signal-processing-based featurization library, a data labeling algorithm – which helps analyze operational data and maintenance events in order to create labels for machine learning problems – and a set of automated machine learning pipelines for predicting outcome types. SMEs incorporate their expertise by providing labeling functions, bands for frequency domain-based featurization, and several other inputs in an intuitive way. We demonstrate the efficacy of this framework through two case studies involving maintenance operation data from wind turbines. Moreover, we show that ML performance can increase when involving domain expertise by a value as high as 48%.

Thesis Supervisor: Kalyan Veeramachaneni
Title: Principal Research Scientist

Acknowledgments

First and foremost, I would like to thank my advisor, Kalyan Veeramachaneni, for his guidance and support throughout the project. His ideas, feedback, and advice were invaluable, and this project would not have been possible without him.

I also wish to acknowledge Sara Pidò and Sarah Alnegheimish for their support and contributions to this project. My thanks goes out to them for their guidance and input on the design and implementation of the framework developed for the project. I would also like to thank everyone in the Data to AI Lab for being incredible labmates and for all the wonderful memories we've shared.

Additionally, I would like to express my gratitude to the Iberdrola team, especially Robert Jones and Sofia Koukoura, for their feedback and insights about the wind energy industry. Without their support, this project would not have been possible.

Finally, I am grateful to my friends and family, especially my parents, Kim and Sam, and my sisters, Austen and Marion, for their constant support and encouragement over the years.

Contents

1 Introduction **15**

1.1 Contributions 17

1.2 Thesis Organization 19

2 Background **21**

2.1 Prediction Engineering 21

2.1.1 Label 21

2.1.2 Segment 22

2.1.3 Featurize 22

2.2 Related Work 23

2.2.1 Prediction of Turbine Faults 23

2.2.2 Wind Data Frameworks 24

2.3 Case Studies 25

2.3.1 Case Study BP: Premature Brake Pad Wear 26

2.3.2 Case Study CF: Converter Failure 27

2.3.3 Challenges 27

3 Zephyr Framework **31**

3.1 Design Goals 31

3.2 Library Overview 32

3.2.1 Use-case Centric Library 32

3.2.2 Framework Overview 33

3.3 Data Representation 34

3.3.1 Signal Data 35
3.3.2 Operations Data 36
3.3.3 EntitySet Representation 36
3.4 Prediction Engineering 38
3.4.1 Writing Labeling Functions 38
3.4.2 Label Search 39
3.5 Feature Engineering 41
3.5.1 Signal Processing 42
3.5.2 Feature Generation 43
3.6 Model Engineering and Evaluation 45

4 Subject Matter Expert Inputs **47**

4.1 Data Input 47
4.2 Label Engineering 48
4.3 Feature Engineering 50
4.3.1 Signal Processing 50
4.3.2 Feature Generation 50
4.4 Model Evaluation 51

5 Evaluation **53**

5.1 Case Studies 53
5.1.1 Setup 53
5.1.2 Brake Pad Case Study 54
5.1.3 Converter Failure Case Study 57
5.1.4 Ablation Study 59
5.1.5 Discussion 60
5.2 Real-World Deployment 61
5.2.1 Addressing Challenges 61

6 Conclusion **65**

6.1 Future Work 66

6.1.1 Library Development 66
6.1.2 Improvement of Prediction Problems 66

List of Figures

1-1 Receiver Operating Characteristic curves for both case studies where blue are results produced by Non SMEs and orange are by SMEs. Overall SMEs improved AUC scores by 48% and 8.3% for the brake pad and converter failure case studies respectively. 18

2-1 An illustration of a turbine drivetrain with six brake pads (in orange, numbered 1-6) located around its circumference. 26

2-2 Part of the electrical system, the converter is responsible for conditioning and managing variable frequency power before it is integrated with the grid. 27

3-1 Zephyr workflow. The process starts with users uploading their data. Zephyr validates and assembles the data by creating an *entityset*, i.e., a structure that organizes the data through relational tables (described in Section 3.3). Users proceed to use Zephyr to create the labeled training examples, which they can iteratively change if they are not satisfied (described in Section 3.4). In the third step, feature engineering, Zephyr allows users to add more variables to the *entityset* to improve model accuracy. Last but not least, the model is built and executed in order to visualize the results. This figure shows the loops within which users can iterate until they are satisfied, both within and between steps. Zephyr natively supports these loops without the user needing to switch to different software packages and/or systems. . . . 33

3-2 Overview of the structure of an *entityset*. The arrows indicate column values in the child dataframe linked to the index of the parent dataframe. T_{ID} is the unique turbine identifier and O_{ID} is the unique work order identifier. 35

3-3 `Python` snippet for creating labeling functions. This function computes the total duration the turbine was stopped for. 40

3-4 This figure depicts how `minimum data, cutoff time, prediction window`, and `gap` can be used to slice data within Zephyr. `Minimum data` is the data left out of the first `prediction window`, as specified by the user. `Prediction window` is the size of the data slice the user wants to consider for their labeling function. This starts at a specific time, namely the `cutoff time`. Finally, the `gap` is the distance between one `cutoff time` and the next. Both the `gap` and the `prediction window` are fixed across all data slices and cannot change. 41

3-5 This figure shows the **feature engineering** step. The first table highlights the parent ID column used as an identifier, the time index columns, and the original features. The aggregation functions are then applied on the original features in order to obtain the second table, which includes the ID column, the time and the engineered features – i.e. the features obtained from the originals after aggregation functions are applied. 43

3-6 XGBoost example for (a) a graph representation of a pipeline (b) a code snippet for training the pipeline using the Zephyr API, then applying inference on the test set. 46

5-1 F1 Scores for the two case studies. Results show the pipelines crafted by SMEs and their respective F1 scores using Zephyr on their Azure platform. 60

List of Tables

2.1 Comparison of ML frameworks. A (✓) indicates the package includes an attribute. Attribute categories from top to bottom: Data used in the framework, either SCADA or PI data; Pipeline denotes the operations handled by each software that includes integration of the data, labeling, featurization, execution of ML model and performance analysis; Framework Characteristics indicates the main features that each software has, automation, customization of labeling and featurization, integration of SMEs knowledge, possibility of creating an end-to-end workflow. 29

3.1 Descriptions of label engineering parameters. 42

3.2 Signal processing transformation primitives available in SigPro. . . . 44

3.3 Signal processing aggregation primitives available in SigPro. 45

5.1 This table reports characteristics of the case study datasets, in particular the number of rows (#Rows), and the number of columns (#Columns). 54

5.2 F1 scores using different lead times. Non SMEs stands for Non Subject Matter Experts, while SME stands for Subject Matter Experts. . . . 63

Chapter 1

Introduction

Wind power is an increasingly popular green energy source. However, because of the harsh conditions in which they operate, wind turbines are susceptible to weather events, bird strikes, corrosion, and other stresses that can cause component failure [18]. Because wind farms are often located in remote or hard to access locations, unexpected component failures lead to high operational and maintenance costs [17]. Predicting and assessing failures in a timely fashion is critical in order to reduce these costs and develop feasible maintenance solutions [12].

There have been several machine learning methods developed to predict failures [23, 24, 10, 5]. However, most developed models are not deployed in the real world, and instead applied only to research studies. Models developed by only ML engineers can be impractical for real-world use because ML engineers lack critical domain knowledge. At the same time, these studies are limited to high profile, high maintenance oriented failures. As wind is becoming a more important source of energy, there is an increasing need to quickly develop practical models at scale for a variety of failures.

Over the past decade, there has been a proliferation of open-source ML tools and systems, along with methods that automate several steps in the development of ML models, including feature engineering, modeling, tuning, and model testing and evaluation. However, building a powerful ML framework that will support a predictive maintenance system remains challenging.

First, it requires flexibility in tackling many prediction problems. These ML frameworks for predictive maintenance systems must be able to adapt to changes in infrastructure or needs such as monitoring new failures, changing prediction windows, or incorporating additional data. There is not yet an easy way to set up a prediction task in the wind energy domain. This can be understood through the lens of data labeling [13]. So far, data and labels for component failure models have been provided a priori, and labels often correspond to major events, such as the failure of a gearbox. Building a model to predict gearbox failure is considered trivial because (1) clearly labeled data that is readily available, and (2) there are sensing systems specifically developed for monitoring the health of a turbine's gearbox [9]. Other predictive problems are more nuanced since failure events and their precursors require deeper search and careful examination.

Second, it needs the integration of domain knowledge at each step of the process. Subject Matter Experts (SMEs) have a deeper understanding of the data, and incorporating their knowledge is vital to generating prediction problems and labels, guiding the generation of new features, and evaluating the practical effectiveness of resulting models. For example, say we want to develop a model that can predict the failure of a particular turbine component: the converter. Expert knowledge is necessary throughout, from prediction task specification through data labeling, featurization, and output assessment. A good example where expert knowledge is required is the data labeling step. Expert knowledge is required to understand whether and when the relevant failure has occurred in the past. For instance, to determine if a converter failure has occurred, an SME would know to search for a work order notification in a time window that contains references to a code indicating converter replacement. Regardless of the specific task, proper labeling often involves processing data from multiple tables, which requires contextual understanding.

Although existing tools are powerful, SMEs often struggle to bridge the gap between generalized, task-specific ML tools, and an end-to-end, domain-specific model. At a practical level, piecing together several tools along with integrating data can result in a significant amount of customized code infrastructure that that has to be

maintained, an issue known as the "pipeline jungle" [22]. Black box solutions may not incorporate domain knowledge, and often result in a lack of trust and understanding. For other tools, the flexibility required to adapt to many different ML tasks leads to complexity and generality that non-machine learning experts may not be able to easily understand and manipulate. The added complexity of knowing which tools to use and how to use them together to generate a model makes it difficult for SMEs to generate models useable in real-world scenarios.

Ideally, a best-of-both-worlds solution allows for automation tools that may lead to an initial solution, while SMEs use their specific knowledge to improve accuracy at every step. In addition to data labeling, this can include adding more data sources or specifying which ones to use, providing feature definitions, and assessing model output. A useful solution will invite the collaboration and trust of SMEs, providing a low-code, flexible framework that enables them to assert their knowledge and influence the process combined with the power of existing ML tools and systems.

1.1 Contributions

To this end, we introduce Zephyr, a domain-specific, data-centric framework for predictive maintenance of wind turbines. Zephyr is a result of a four-year collaboration [15] between system builders and subject matter experts. Figure 1-1 is a preview of Zephyr's ability to improve model performance via SME involvement for the initial case studies investigating brake pad and converter failure[1] (Chapter 5). When SMEs participated in data label identification and feature engineering, the AUROC improved by 48% and 8.3%. More importantly, involvement at every stage of the development process allows subject matter experts to understand and trust the resulting model. We designed the Zephyr ecosystem to be composable, such that SMEs can inject domain-specific expertise into each step of the framework.

To summarize, the contributions presented in this thesis are as follows:

- The creation of Zephyr, a data-centric framework that spans the entire scope

[1] These studies are still in progress at the time of writing this thesis.

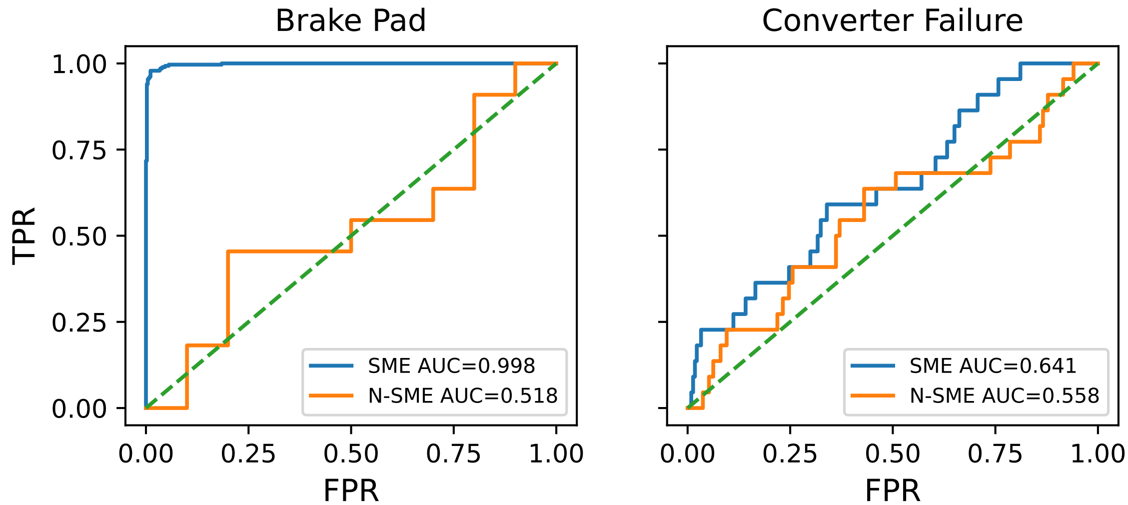


Figure 1-1: Receiver Operating Characteristic curves for both case studies where blue are results produced by Non SMEs and orange are by SMEs. Overall SMEs improved AUC scores by 48% and 8.3% for the brake pad and converter failure case studies respectively.

of data to model development, replacing the model-centric view, which focused more on models rather than the end-to-end process. Zephyr focuses on multi-source data aggregation, data labeling and feature engineering, and transfers well to production.

- A framework designed to enable SMEs to incorporate their knowledge at each stage, improving the resulting models and enhancing SME understanding and trust.
- Evaluation of Zephyr through two real-world case studies demonstrating how performance is improved with SME input. The case studies were performed and evaluated in collaboration with Sara Pidò.

To ground the framework and its applicability, the case studies informed where customization was necessary leading to some of the design choices for Zephyr. A working paper, titled "Sintel Zephyr: a Data-Centric Framework for Predictive Maintenance of Wind Turbines", is in progress that covers the Zephyr framework. It is

written with equal contributions from Sara Pidò and Sarah Alnegheimish and is currently under review. Additionally, SMEs and data scientists Sofia Koukoura and Robert Jones' feedback and help drive the case studies and the design choices in the framework.

1.2 Thesis Organization

This thesis is organized as follows: Chapter 2 introduces important technical concepts, presents related work in predictive maintenance of wind turbines, and introduces two real-world case studies that were used to evaluate the framework. Chapter 3 presents the overall design of the Zephyr framework, as well as each individual stage from the integration of data sources to the execution of the machine learning model and visualization of the results. Chapter 4 details how the framework is designed to take in subject matter expert input and the tunable controls available to the user. In Chapter 5, we present and evaluate the results obtained from two real-world case studies. Finally, Chapter 6 summarizes the conclusions and details future work on the project.

Chapter 2

Background

2.1 Prediction Engineering

Prediction engineering describes the process of going from raw timestamped data to feature vectors and associated labels [13]. This thesis borrows from the Label-Segment-Featurize framework created by Kanter, et al. [13] which separates prediction engineering into three primary tasks: labeling, segmentation, and featurization. The work in this thesis uses the Label-Segment-Featurize abstraction and specifically applies it to wind farm data.

2.1.1 Label

The labeling step searches through the data to generate labeled samples for slices of data. Each sample is also mapped to a **cutoff time** which is the initial timestamp associated with the corresponding data slice. The result from the labeling step are **label times**, three element tuples consisting of the cutoff time, unique target entity, and the label. In machine learning, the label times tell the algorithm when an event or outcome we are attempting to predict happened.

The target entity is the real world thing we want to make predictions for, for example an individual wind turbine. In this thesis, the target entity is generally a unique wind turbine since prediction problems are often turbine focused.

The cutoff time is the time associated with the label. Cutoff times are crucial to prevent data leakage because any data after the cutoff time is directly used to generate the label for the data slice. All data before the cutoff time is considered valid for use as training data for the corresponding label time, while all data after the cutoff time is ignored.

The label represents the problem we are trying to solve and is the target we are trying to predict. The type of a label is dependent on the problem, for example, if we are trying to predict if a turbine will fail a label will be either `True` to indicate a failure occured or `False` otherwise. Labels can also be multiclass or numerical values.

2.1.2 Segment

Next, the segmentation step generates data useable for learning for each sample generated during the labeling step. While the cutoff time provides the bare minimum to prevent data leakage, it is often beneficial to further restrict the data useable for learning. For example, if it is desired to predict `n` days in advance, then data useable for learning should not include the `n` days prior to the cutoff time for a sample. This is known as applying a **lead** to the label time. Segmentation can be applied directly to a label time's cutoff time to change what is considered valid learning data for that specific label time. For example, shifting a cutoff time backwards by one day prevents data from that day being used in training, meaning the model will predict one day in advance.

2.1.3 Featurize

Finally, feature engineering is applied to the valid learning data for each sample to generate a feature vector that can be used for machine learning. Feature engineering is critical to the performance of machine learning models. Feature engineering changes how the raw data is represented to a format that is more meaningful to a machine learning model [14]. At its core, feature engineering relies on applying mathematical operations to data to either transform or aggregate data into more useful values.

Traditional feature engineering was done through hand-crafted features made by subject matter experts, although recently automated feature engineering algorithms have shown significant improvements in model performance as well [14]. In this work, traditional signal processing techniques are combined with feature engineering to generate rich feature vectors.

2.2 Related Work

Wind turbine profitability is critical for energy companies investing in this green technology, and wind farms seek to improve their operational performance while reducing operation and maintenance costs. In this context, many machine learning tools and frameworks have been developed in order to predict failures and optimize maintenance interventions. As shown by [21], machine learning developments and their priorities are determined through a systematic process called "Failure Mode Effects and Criticality Analysis" (FMECA), and then extended through an analysis of the respective failure mechanisms and monitoring options.

2.2.1 Prediction of Turbine Faults

Much of the current work on wind turbines focuses only on predicting mechanical and electrical failures to avoid excessive downtime and improve prevention and predictive maintenance.

Some studies focus on detecting anomalous behaviour in a turbine's performance in order to uncover faults. For example, [23] showed that deviations in the mean, baseline and Kurtosis of a baseline compared to online values of the power curve highlight performance improvements after corrective maintenance actions. In another work, [24] used a correlation metric between various turbine parameters (e.g. wind speed and power) to compare fault-free values to live values and detect faults. They demonstrated that this measure reliably detects mechanical degradation before wind turbine blade and drive train failures.

Others focus their attention on building classification models to determine whether

a turbine is operating normally, or if a fault has happened or will happen in the near future. For example, [10] applied Random Forests and Decision Trees to build predictive models for turbine failures, and proved that machine learning helps to make these predictions. [5] showed that faults could be predicted with high precision and accuracy using an Adaptive Neuro-Fuzzy Inference System.

These works focus largely on prediction and, in particular, studying and developing model-centric machine learning solutions. Model-centric machine learning solutions tend to focus exclusively on improving performance by modifying the model itself, as opposed to data-centric methods which approach the data to model problem more holistically. Model-centric techniques neglect the importance of data to the performance of the machine learning models and, as a result, may suffer from worse performance and be more opaque to a subject matter expert compared to a data-centric model.

More recently, new data-centric solutions have been developed. These methods provide pipelines that may include data preprocessing, feature engineering, and feature selection, as well as model building and training. For example, [8] developed a data-centric ML pipeline focusing on the optimization of the preprocessing and feature selection steps. However, they do not provide a complete end-to-end pipeline that performs all of these steps and executes the model.

2.2.2 Wind Data Frameworks

The community has continued to develop frameworks that use wind turbine data to predict faults. One example is by Nguyen Et al. [19] who developed a framework for the integration of wind turbine data in order to optimize the remote operations of offshore wind farms. Jiang Et al. [11] implemented a deep learning framework which allows multiple wind turbines to collaboratively build a fault detection model using their private data. Additionally, Leahy Et al. [16] developed a framework that starts from wind turbine and alarm data to build a model that can classify normal versus faulty turbines. They created batches of alarm data, cleaned and labeled the data, and built a classification model. Finally, they measured the performance and

evaluated the model.

All these methods are focused on predicting faults in wind turbines. While one integrates wind turbine data [19], to the best of our knowledge, a framework has not been developed that performs the whole workflow. Table 2.1 shows the comparison between the Zephyr framework and existing frameworks, both for doing wind turbine prediction analysis and more general machine learning framework. The Zephyr framework allows users to start from different kinds of data, join and integrate them, create labels for a prediction problem, and build features in order to construct a machine learning model. Furthermore, our framework does not impose a particular labeling function on the users; instead, they can choose between a set of predefined labeling functions or their own custom function and answer questions that go beyond failure presence.

While to our knowledge no end-to-end frameworks have been developed for the renewable energy domain, other end-to-end, domain specific frameworks have been successfully developed. One such example is Cardea [4], a flexible, end-to-end machine learning framework for the healthcare domain.

2.3 Case Studies

Zephyr has been created in partnership with a team of SMEs from a wind energy company who have provided key insights as well as a significant amount of data. The SME team works with both on- and offshore wind turbines that demand continuous monitoring. Because predictive maintenance can reduce the costs induced by failures and help in planning future repairs, they wish to use automated tools and machine learning methods to predict when failures might occur. Our objective is to create a framework for predictive maintenance of wind turbines that enables SMEs to easily and intuitively incorporate their domain expertise. To showcase Zephyr's ability to integrate specific domain knowledge, we present case studies for two different failure types: premature brake pad wear and converter failure.

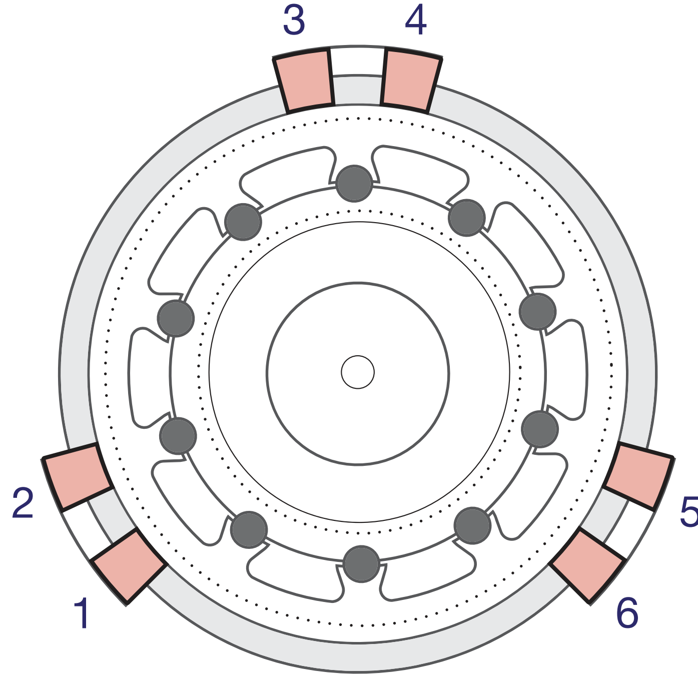


Figure 2-1: An illustration of a turbine drivetrain with six brake pads (in orange, numbered 1-6) located around its circumference.

2.3.1 Case Study BP: Premature Brake Pad Wear

The first case study looks at **B**rake **P**ad (**BP**) wear at Wind Farm 1. This wind farm has 70 turbines, for which 55% of brake pads were replaced in 2021. Figure 2-1 is a high-level depiction of the brake pads present in a turbine. Brake pads on the mechanical drivetrain become worn with use, and are designed to wear out within an expected time that is aligned with maintenance windows. However, premature wearing of brake pads often necessitates their replacement before scheduled maintenance. Pad replacement can take up to 5-6 shifts to be completed; in addition, current efficient management of brake pad replacement requires taking periodic physical measurements of the pads, which is costly and can be risky for maintenance personnel. Moreover, because of the high volume of brake pad wear occurring outside of the expected maintenance windows, personnel are required to physically climb up the turbine and check the pad. This engagement is risky and expensive. Having a data-driven monitoring approach can help with planning and prioritizing checkups, ideally reducing the cost and risk.

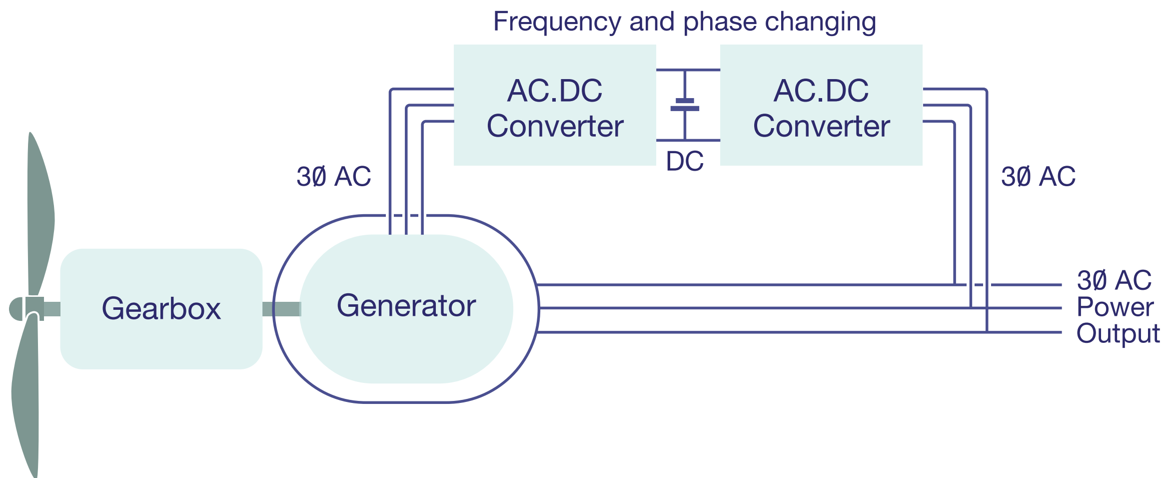


Figure 2-2: Part of the electrical system, the converter is responsible for conditioning and managing variable frequency power before it is integrated with the grid.

2.3.2 Case Study CF: Converter Failure

The second case study examines a **C**onverter **F**ailure (**CF**) problem at Wind Farm 2. This wind farm includes over 150 turbines. A critical part of the electrical drivetrain, the converter, is responsible for efficiently managing and conditioning the variable frequency power output from the generator prior to local grid integration at the expected frequency. Figure 2-2 provides a general overview of converters in wind turbines. A converter comprises a range of power electronics, any of which can suffer from a variety of unexpected failures [7] and cause the turbine to stop operating. In Wind Farm 2, more than 60% of turbines have experienced at least one converter failure. Converters are modular, and parts can typically be replaced quickly during an up-tower repair to avoid prolonged outages. Unlike other turbine components, converters do not usually have their own custom monitoring systems, meaning an ML-based monitoring approach using available data would be useful.

2.3.3 Challenges

For both of these problems, traditional rule-based methods that combine statistical indicators are currently used to create alerts and notify operation and maintenance teams of any malfunctions. However, these alerts often do not provide enough advance

warning for planning and maintenance action, and can even miss failures entirely. Introducing machine learning can facilitate predictive maintenance for each of these components. However, there are several challenges.

First, a vast amount of assets and signals must be monitored, necessitating pulling together data from different places and in different formats.

Second, the raw data provided often lacks the right context and labels for machine learning model building. In some cases, there are not enough failure examples or reliable maintenance records to create an accurate model.

Finally, although SMEs can provide the necessary data and context, they may find it difficult to work with machine learning tools. With Zephyr, we provide an automated and scalable framework for creating machine learning prediction problems and building ML models to predict component failure. Zephyr unifies various data types collected from different data sources, and provides flexibility in generating labels, customizing existing problems, and creating new ones. Zephyr allows SMEs to interact productively with the framework through a simple and intuitive interface (APIs).

		WTPHM [16]	DeepFedWT [11]	AWS Fleet Predictive Maintenance	Human First AI [1]	Pycaret [3]	**Zephyr**
Data	SCADA data	✓					✓
	PI data						✓
Pipeline	Integration of the Data	✓	✓				✓
	Labeling	✓					✓
	Featurizaton			✓	✓	✓	✓
	Execution of ML Model		✓	✓	✓	✓	✓
	Performance Analysis		✓	✓	✓	✓	✓
Framework characteristics	Automation	✓	✓				✓
	Customizable Labeling Function	✓					✓
	Customizable Featurization			✓		✓	✓
	Integration of SMEs Knowledge	✓		✓	✓	✓	✓
	End-to-end workflow	✓	✓	✓	✓	✓	✓

Table 2.1: Comparison of ML frameworks. A (✓) indicates the package includes an attribute. Attribute categories from top to bottom: Data used in the framework, either SCADA or PI data; Pipeline denotes the operations handled by each software that includes integration of the data, labeling, featurization, execution of ML model and performance analysis; Framework Characteristics indicates the main features that each software has, automation, customization of labeling and featurization, integration of SMEs knowledge, possibility of creating an end-to-end workflow.

Chapter 3

Zephyr Framework

The Zephyr[1] framework has been developed in order to help SMEs generate machine learning pipelines for wind farm data, providing them the flexibility and extensibility required to add domain-specific knowledge. The library is used to ingest standardized operations data from wind farms, generate labels for past events for a given prediction problem, and facilitate feature generation for the purposes of creating a machine learning model. Zephyr focuses on abstraction and automation to facilitate the end to end process of labeling, feature generation, and modeling. Zephyr's design also focuses on user input through tunable parameters, customized inputs, and extensibility options to enable SME knowledge to guide the entire process. This chapter introduces the Zephyr framework and each component, as well as how SME input is leveraged throughout the process. This chapter will refer to SMEs as "users" in order to underline their position as the primary users of the framework.

3.1 Design Goals

The main goal of the Zephyr framework is to allow SMEs to easily go from wind farm data to a machine learning model. Thus, much of development was focused on useability and ensuring that Zephyr is flexible enough to meet the needs of its

[1]meaning a gentle, mild breeze and coming from the name of the Greek god of the west wind, Zephyrus.

users. The framework should provide basic building blocks and guidance and enable SMEs to leverage their domain knowledge to customize and enhance the process. This includes making it simple for SMEs to define new prediction problems, change input parameters, and evaluate resulting models. Zephyr should also be able to adapt to the changing needs and environment of its users. The framework requires flexibility at each stage in order to account for new or different data, new prediction problems, or additional knowledge available to the SMEs. Flexibility is also critical to enable experimentation and iterative development: experimenting with parameters should not require extensive changes to continue developing new models.

3.2 Library Overview

The Zephyr library is an open-source Python package hosted on GitHub and publicly available through PyPI. This section provides an overview of the overall design choices made when creating the library.

3.2.1 Use-case Centric Library

Unlike a tool based library designed to perform a task across many domains, Zephyr is a use-case centered, domain-specific library. It is designed to encapsulate the end-to-end workflow of developing a model for the renewable energy domain for use-cases defined by the available prediction problems. As a result, Zephyr acts as a wrapper around multiple general-purpose libraries such as `Featuretools`[2] and `Compose`[3], which themselves are based on foundational libraries such as `pandas`[4] and `NumPy`[5].

While ML-task and general-purpose libraries are powerful, we discovered while working with our SME partners that they are often too generalized and difficult to compose together into an entire pipeline that can be used in a real-world setting. General-purpose libraries often have flexible APIs with many available parameters to

[2]https://github.com/alteryx/featuretools
[3]https://github.com/alteryx/compose
[4]https://pandas.pydata.org/
[5]https://numpy.org/

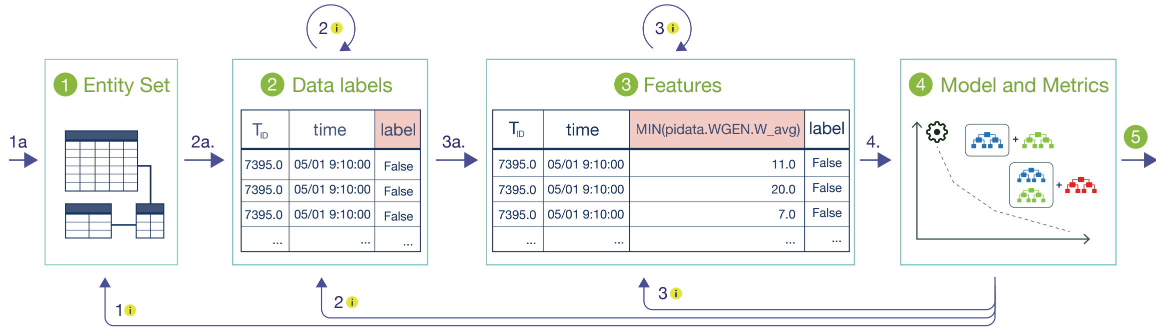


Figure 3-1: Zephyr workflow. The process starts with users uploading their data. Zephyr validates and assembles the data by creating an *entityset*, i.e., a structure that organizes the data through relational tables (described in Section 3.3). Users proceed to use Zephyr to create the labeled training examples, which they can iteratively change if they are not satisfied (described in Section 3.4). In the third step, feature engineering, Zephyr allows users to add more variables to the *entityset* to improve model accuracy. Last but not least, the model is built and executed in order to visualize the results. This figure shows the loops within which users can iterate until they are satisfied, both within and between steps. Zephyr natively supports these loops without the user needing to switch to different software packages and/or systems.

adapt to many use cases, but this can increase complexity and confusion in usage. Many tools are built for a single task and have strict input and output specifications, which can make it difficult for users to use multiple tools in a single pipeline. As a result, Zephyr was developed to create a framework that uses these underlying general ML tool libraries, but is designed specifically for the wind turbine domain. Re-framing machine learning concepts in a domain-specific context helps the SMEs better understand what each step in the framework is designed to do which, in turn, allows them to more easily leverage their domain knowledge in the process. Additionally, encapsulating general purpose libraries with convenience wrappers in Zephyr helps simplify usage of underlying libraries for the SMEs by providing domain-specific defaults, removing unnecessary features, and simplifying the API.

3.2.2 Framework Overview

Zephyr is an end-to-end, data-centric framework for generating machine learning models. Beginning with data ingestion, users load their datasets into the framework, which

assembles them into a usable standardized format. Once data has been loaded, users generate data labels from it by defining their own labeling function or selecting one from a predefined list of prediction problems. Next, the automated machine learning phase of the framework starts, which consists of three primary steps: feature engineering, model selection, and hyperparameter tuning. Finally, the framework helps a user analyze their model. By breaking down model generation into these principle steps, Zephyr simplifies and standardizes the process, enabling structured iterative improvements and reducing complexity.

Figure 3-1 depicts the Zephyr workflow, highlighting the inherent loops in the development process. Each stage of the framework is designed to take in user knowledge. Users can also go back and change choices they initially made to experiment with different configurations. For example, let's consider the converter failure case study. The user started by uploading required tables and executing the labeling phase of the framework. After that, the user created some new features and trained the model. The obtained Area Under the Curve (AUC) was close to random — around 0.5. Thus, the user went back to improve the model by changing and adding features.

The remainder of this chapter describes each component of the framework.

3.3 Data Representation

The first step in the Zephyr framework is to combine available signal data and operations data into a well-formatted, relational data structure. Much of this data is standard across wind turbines and farms, which we use to our advantage when ingesting data. It is critical for Zephyr to load data following the raw/natural format that it is stored in. This allows for continuous data/model updates as more data becomes available [20]. Each of the two main data sources, signals data and operations data, is described in more detail below.

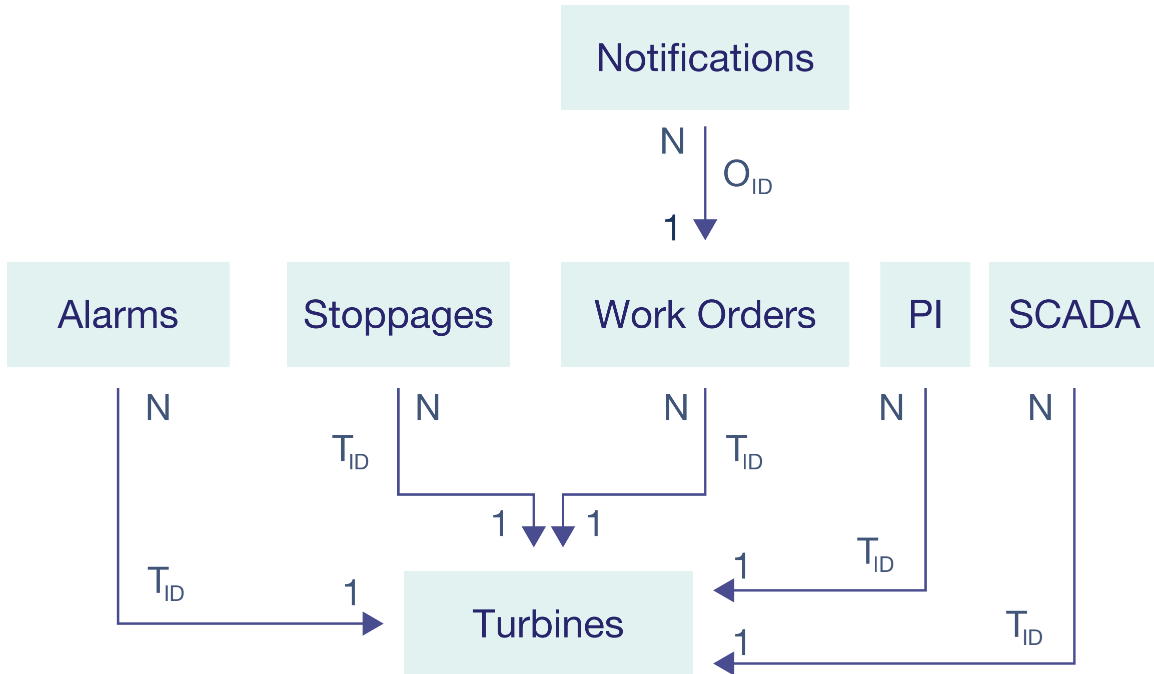


Figure 3-2: Overview of the structure of an *entityset*. The arrows indicate column values in the child dataframe linked to the index of the parent dataframe. T_{ID} is the unique turbine identifier and O_{ID} is the unique work order identifier.

3.3.1 Signal Data

Signal data is time series data that originates from wind turbine assets and their auxiliary systems. There are two different standard data sources: the Original Equipment Manufacturer Supervisory Control And Data Acquisition (OEM-SCADA) system, and the operator's chosen historical Plant Information (PI) system. We observe that there may be overlaps or redundancies between these two signal data sources, although typically one is more information-rich in terms of the number of monitored signals and/or data collection frequency. Specific data collected at each wind farm could be a subset of the overall set of signals we incorporate in Zephyr. Particularly, PI/SCADA signals are sampled at regular intervals as specified in the query (for example, every 10 min). At each timestamp, this data contains numerical information on different types of wind turbine variables/signals, such as rotor speed, active power, current, and voltage, and is aggregated using a variety of functions – minimum, maximum, average, and standard deviation – for each interval.

3.3.2 Operations Data

Operations data is collected during day-to-day operations and maintenance of a wind farm. Much of this data is standardized as it comes from SAP Application Performance Standard (SAPS) [2] and other systems. Operations data includes:

- Alarms: Alarms generated by onboard diagnostic systems in the turbines and corresponding timestamps. Each alarm occurrence usually has a time period (start-end) and an alarm description.
- Stoppages: Information about stoppages associated with each turbine such as turbine ID, stoppage start time, stoppage end time, and cause description.
- Work orders and contextual information: Maintenance history information for each turbine, including order ID, order creation date, functional location, and other attributes.
- Notifications: Information regarding maintenance history on top of work orders, including attributes like malfunction start timestamp, malfunction end timestamp, fault mode description, and what parts were replaced or repaired.

3.3.3 EntitySet Representation

To organize this data into relational tables, Zephyr uses *entitysets* [14]. An *entityset* is an in-memory representation of a collection of tables and the relationships between them. Each table (referred to as an *entity*) in an *entityset* contains metadata about columns and the associated data types, and allows columns to be marked as time indexes for data rows. A `time index` is a timestamp column in a table that defines the point in time at which the data in the row was known. Multiple `secondary time index` columns may also be specified, which are timestamp columns that map to the time when the data in specific columns in the row was known. Secondary time indices are often used for data that is added to a row after the time index. For example, a work order may have a `time index` of when the work order was created and a `secondary time index` of when the work order is closed for a parts consumed

column that only gets updated after a work order has been completed. The *entityset* representation has been successfully applied in other domains, such as healthcare [4].

Categorical columns in an *entityset* can also have additional metadata in the form of `interesting values`. These are the specific categorical values that are known by SMEs to have more significance, and are used during the feature generation step described in Section 3.5.2. For example, if a specific alarm is known to be a common indicator of certain conditions, users can mark the alarm code as a interesting value in the `alarms` entity which will allow the feature generation step to create features specific to instances of that alarm code.

A primary advantage of the *entityset* representation is that it allows for users to go from raw tables to a single standardized relational structure. Once data has been ingested and converted to an *entityset*, every downstream step can take advantage of the metadata contained in the *entityset* structure. Furthermore, an *entityset* can easily be serialized to disk which enables data and workflow sharing, enhancing collaboration opportunities. Directly sharing an *entityset* also reduces reproducibility errors caused by ingesting and formatting raw data, such as type mismatches or type conversion errors.

The structure of the *entityset* generated for the above data sources is depicted in Figure 3-2. A turbine table is central to the structure and includes a unique identifier for all turbines, shown as T_{ID}. Other tables have a foreign key relationship based on this `turbine ID`. The `notifications` and `work orders` are connected via the unique work order identifier, shown as O_{ID}. We chose *entityset* as a Python representation for the data because it maintains the relational structure of data during labeling and featurization. Moreover, it enables us to perform data validation to verify that the data can be ingested by checking for mandatory columns, such as parent-and-child relationships and time indices. *entityset*s form a cornerstone in our automation because the rest of the framework is able to work with these representations.

When attempting to use Zephyr across wind farms, one common occurrence is that different farms have different subsets of entities or columns. The *entityset* creation step is designed to be as flexible as possible to allow users to add or remove

tables, columns, or rows while still maintaining the overall structural integrity of the *entityset*. Most data columns can be removed or additional columns can be added to individual *entities*; the framework requires only index columns and columns used as foriegn keys to other *entities*. Additional tables can be added as separate *entities* to the *entityset*, or *entities* can be removed if the data is irrelevant or unavailable. For example, if additional signals or operation data is available, a new table can be added to the *entityset* simply by adding the table and defining any relationships between the new table and the existing structure, e.g. mapping a turbine index column to the parent turbine entity. *Entities* can also be removed entirely from the *entityset* if needed, although it is generally preferable to leave *entities* for data integrity and instead ignore unwanted entities in later steps.

Another possible scenario involves different farms using different names for a subset of fields across *entities*. With standardization, this is becoming less and less common. However, Zephyr provides flexibility by enabling the user to provide a map for column names in the standardized *entityset* and equivalent names in their data.

While the expected data columns have specified data types, data types of additional columns will be automatically inferred unless the column type is explicitly set during entity creation [14]. Columns can also be cast to new types after the *entityset* has been created.

3.4 Prediction Engineering

Once an *entityset* has been created, the user moves on to defining their prediction problem and generating labeled training examples from their data. This process involves two components: defining a *labeling function* and performing a *label search* over the data to generate label times.

3.4.1 Writing Labeling Functions

Zephyr offers a selection of predefined prediction problems including: predicting the stoppage of a wind turbine because of a brake pad issue, predicting converter replace-

ment, and predicting the total power lost over a period of time. These problems are defined through a specific function called a *labeling function.*

The framework is also built to be easily extensible through the creation of custom labeling functions. Users can create additional labeling functions by: (1) defining how a label is constructed for one particular data slice; (2) defining how the *entityset* should be denormalized into a single table to generate each data slice; and (3) setting default label search parameters such as `gap` or number of labeled training examples per turbine. The label search parameters are discussed in more detail in Section 3.4.2.

In Figure 3-3, we demonstrate how an additional labeling function can be created. First, a new labeling function is defined. In this case, we want to label each of our data slices with the total amount of time for which the turbine was stopped during the window. Next, we define some metadata for our `labeling function`, namely that our `target entity index` is the `T_ID` and that the `time index` we want to use is the `END_DATE`. Finally, we determine how we want to denormalize our *entityset* into a single dataframe for generating slices. Denormalization is done through a helper function which sequentially merges the*entity* tables in order. In this example, we only need the `stoppages` *entity* from our *entityset*, but multiple *entities* can be listed and flattened into a single dataframe. Using a few lines of code, users can create new prediction problems.

3.4.2 Label Search

To generate labeled training examples, Zephyr performs a search process over the relevant data. Zephyr uses methods from the open source library `Compose`[6], which focuses on tools for automating the prediction engineering process, to help generate label times. `Compose` is used to generate data slices and apply labeling functions using the label search parameters passed to the prediction engineering process. Each row in the target table is segmented based on its associated `cutoff time` into: (1) data before the cutoff time, which is used for training (2) data after the cutoff time, which is used to generate labels. The amount of data used to generate a label is dictated by

[6]https://github.com/alteryx/compose

```python
def total_stoppage_time(es):
    """ Determines the total stoppage time
        for a turbine during the window """
    def label(ds):
        return ds['DURATION'].sum()

    meta = {
        "target_entity_index" = "T_ID",
        "time_index" = "END_DATE"
    }

    df = denormalize(es, entities=["stoppages"])

    return label, df, meta
```

Figure 3-3: `Python` snippet for creating labeling functions. This function computes the total duration the turbine was stopped for.

the `prediction window`, which sets the size of each data slice that will be taken into consideration when generating labels. The beginning of the prediction window aligns with the corresponding cutoff time, and the time between one cutoff time and the next cutoff time is the `gap`. By default, the first cutoff time occurs at the beginning of the data; however, users can optionally specify the `minimum data` required before the first cutoff time, shifting the first cutoff time forward by that amount. Figure 3-4 demonstrates how each of these parameters influences the label search.

Additionally, users can apply a `lead` to the cutoff times to shift them by the given amount. This shifts the data window allowed to train the model to account for the amount of time the user wants a prediction in advance. The zoom-in section of Figure 3-4 highlights how the lead is represented. For example, a `lead` time of 2 days would mean that any data from up to 2 days before the labeled event would no longer be available for training.

Finally, users can also control the maximum number of labeled examples they wish to generate per instance. This is often useful when a large number of labels may be present or when experimenting with different parameters, as it can reduce the time needed to execute the label search.

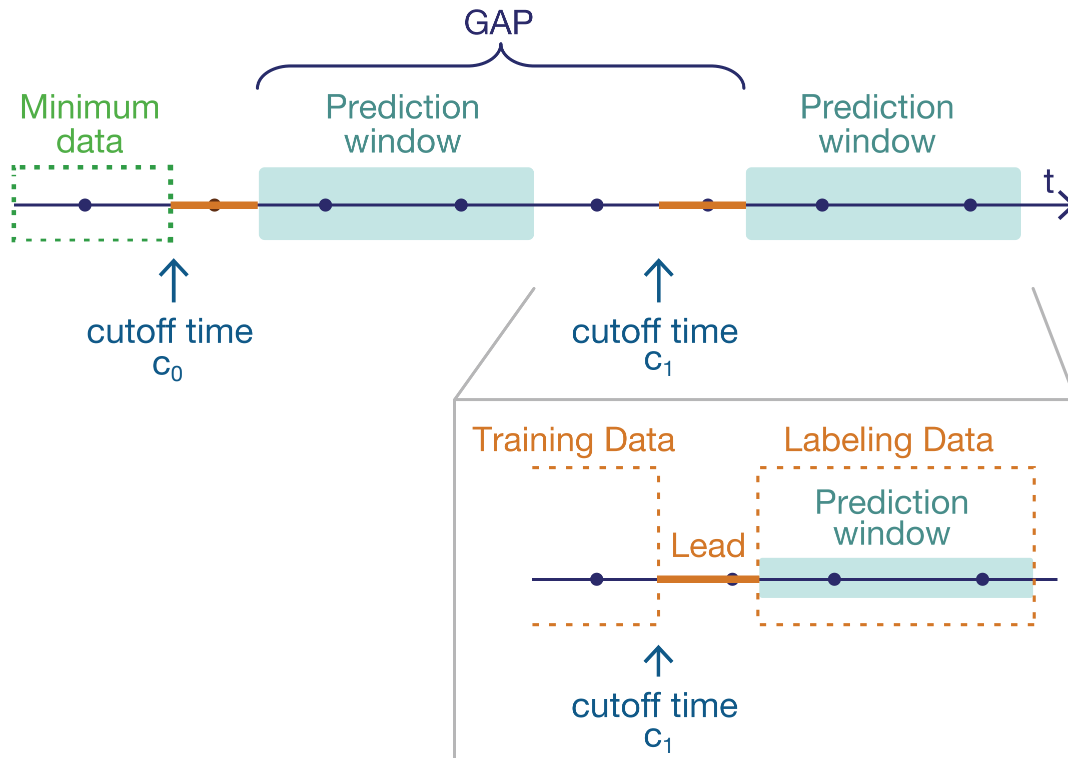


Figure 3-4: This figure depicts how `minimum data`, `cutoff time`, `prediction window`, and `gap` can be used to slice data within Zephyr. `Minimum data` is the data left out of the first `prediction window`, as specified by the user. `Prediction window` is the size of the data slice the user wants to consider for their labeling function. This starts at a specific time, namely the `cutoff time`. Finally, the `gap` is the distance between one `cutoff time` and the next. Both the `gap` and the `prediction window` are fixed across all data slices and cannot change.

These prediction engineering parameters allows users to fine-tune the search for labeled training examples based on their data and prediction needs. While this Section has explained the most important parameters, the full list of available parameters to control the label engineering process is summarized in 3.1.

3.5 Feature Engineering

As the next step in an ML workflow, relevant features need to be extracted from the data in order to improve the performance of a model. To improve the model's accuracy, Zephyr enables the creation of features to transform the *entityset* structure into a feature matrix that can be used to train a model. Zephyr can use a combination of

Parameter	Description
`window_size`	The size of prediction window.
`num_samples`	The number of label samples to return for each unique target entity.
`minimum_data`	The minimum amount of data or time before starting the label search process.
`maximum_data`	The maximum amount of data to process before stopping the label search process.
`drop_empty`	Whether or not to drop empty slices of data when searching for labels.
`gap`	The amount of time between consecutive samples.
`lead`	The amount of time to shift cutoff times by to create an advanced prediction window.
`threshold`	An optional minimum threshold value to use to convert continuous labels to binary. The label is `True` if the value exceeds the provided threshold value
`column_map`	An optional dictionary used to change the default mapping of column aliases to column names used by the dataframe.

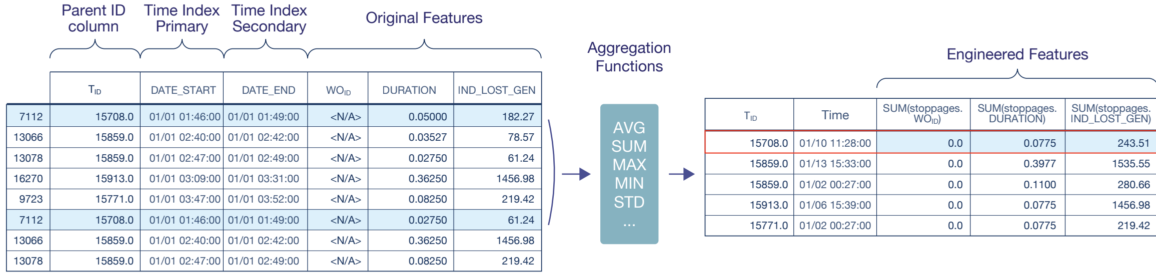
Table 3.1: Descriptions of label engineering parameters.

signal processing techniques and automatic relational data manipulations for feature engineering.

3.5.1 Signal Processing

Signal data is transformed in Zephyr using `SigPro`[7], a signal processing library co-developed for use with Zephyr. After labeling, users can choose to further process time series signal data using common signal transformation and aggregation functions. Users select the table(s) containing signal data to be processed, indicate which column(s) contain signal data, specify the time window to apply aggregations over, and specify a signal processing pipeline to use. Signal processing pipelines are defined by selecting which transformation and aggregation primitives to apply and selecting the appropriate hyperparameters for them. Transformations are applied sequentially

[7]https://github.com/sintel-dev/SigPro



	T_{ID}	DATE_START	DATE_END	WO_{ID}	DURATION	IND_LOST_GEN
7112	15708.0	01/01 01:46:00	01/01 01:49:00	<N/A>	0.05000	182.27
13066	15859.0	01/01 02:40:00	01/01 02:42:00	<N/A>	0.03527	78.57
13078	15859.0	01/01 02:47:00	01/01 02:49:00	<N/A>	0.02750	61.24
16270	15913.0	01/01 03:09:00	01/01 03:31:00	<N/A>	0.36250	1456.98
9723	15771.0	01/01 03:47:00	01/01 03:52:00	<N/A>	0.08250	219.42
7112	15708.0	01/01 01:46:00	01/01 01:49:00	<N/A>	0.02750	61.24
13066	15859.0	01/01 02:40:00	01/01 02:42:00	<N/A>	0.36250	1456.98
13078	15859.0	01/01 02:47:00	01/01 02:49:00	<N/A>	0.08250	219.42

T_{ID}	Time	SUM(stoppages. WO_{ID})	SUM(stoppages. DURATION)	SUM(stoppages. IND_LOST_GEN)
15708.0	01/10 11:28:00	0.0	0.0775	243.51
15859.0	01/13 15:33:00	0.0	0.3977	1535.55
15859.0	01/02 00:27:00	0.0	0.1100	280.66
15913.0	01/06 15:39:00	0.0	0.0775	1456.98
15771.0	01/02 00:27:00	0.0	0.0775	219.42

Figure 3-5: This figure shows the **feature engineering** step. The first table highlights the parent ID column used as an identifier, the time index columns, and the original features. The aggregation functions are then applied on the original features in order to obtain the second table, which includes the ID column, the time and the engineered features – i.e. the features obtained from the originals after aggregation functions are applied.

and each aggregation is applied individually to the transformed signals data, resulting in a single value for each aggregation. Once processed, these signals can either replace the original signal's *entity*, or be added as an additional *entity* while retaining the original signal table. SigPro supports many common signal processing techniques with customizable hyperparameters, including signal transformations such as Fourier transforms, as well as aggregations over the whole signal or specific frequency bands or side bands. A summary of the available signal processing primitives is shown in 3.2 and 3.3.

3.5.2 Feature Generation

Additional features that exploit the relational structure present in the *entityset* can be automatically generated using `Featuretools` [14]. `Featuretools` recursively applies data manipulation primitives that transform data columns or aggregate columns across table relationships defined by the *entityset* format. `Featuretools` is also applied over any time series features generated with `SigPro` to leverage both common signal processing with enhanced feature generation. Because `Featuretools` also uses *entityset*s to work with the data, Zephyr seamlessly integrates with the `Featuretools` library and exposes essential arguments for users.

Users can select what type of `aggregation` and `transformation` functions should

Primitive	**Type**	**Description**
`identity`	Transformation	The identity transform
`shift_frequency`	Transformation	Shifts the frequency based on the current RPM versus the given nominal.
`power_spectrum`	Transformation	Generates the power spectrum.
`envelopespectrum`	Transformation	Computes the envelope spectrum.
`frequency_band`	Transformation	Extracts a specific specific band by filtering between a high and low band frequency.
`fft`	Transformation	Apply a discrete Fourier Transform on the amplitude values.
`fft_real`	Transformation	Apply a discrete Fourier Transform on the amplitude values and return the real components
`stft`	Transformation	Compute the Short Time Fourier Transform
`stft_real`	Transformation	Compute the Short Time Fourier Transform and return the real components

Table 3.2: Signal processing transformation primitives available in SigPro.

be applied, `drop tables` and `columns` which they think would not lead to valuable features, specify interesting values in the data, and insert seed features. Interesting values are set for categorical columns on an *entity* and are used to indicate specific values to filter on when generating features. For example, a user may mark a specific alarm as interesting so as to generate features specific to that alarm in each turbine.

Seed features are manually defined features that are designed to allow users to specify known useful features that should be generated and used during automated feature generation. Seed features ensure that known important features are generated and used during the feature generation process, and rely on domain knowledge to be created. Seed features and their importance are discussed in more detail in Section 4.3.

The cutoff times generated by the labeling step described in 3.4 are used to ensure that feature generation does not result in data leakage. During the feature generation process, any data from after the cutoff time is ignored when creating a feature.

Primitive	**Type**	**Description**
`crest_factor`	Aggregation	Computes the ratio of the peak to the root mean square
`kurtosis`	Aggregation	Computes the kurtosis
`mean`	Aggregation	Calculates the mean value of the values
`rms`	Aggregation	Computes the root mean square
`skew`	Aggregation	Computes the sample skewness
`std`	Aggregation	Computes the standard deviation value
`var`	Aggregation	Computes the variance value
`band_max`	Aggregation	Computes the maximum values for a specific band
`band_mean`	Aggregation	Computes the mean values for a specific band
`band_min`	Aggregation	Computes the minimum values for a specific band
`band_rms`	Aggregation	Computes the root mean square value for a specific band
`band_sideband_pr`	Aggregation	Computes the power ratio values for side bands versus a specific band
`band_sideband_rms`	Aggregation	Computes the root mean square values for a specific band and associated sidebands
`band_sum`	Aggregation	Computes the sum of values for a specific band

Table 3.3: Signal processing aggregation primitives available in SigPro.

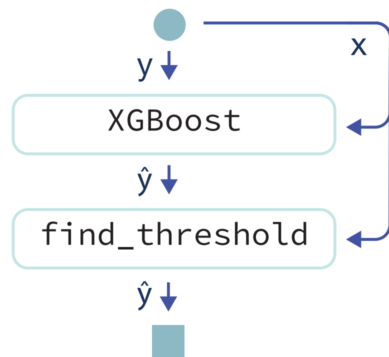
3.6 Model Engineering and Evaluation

Considering the large number of machine learning libraries, each with a variety of models that can be applied on featurized data, we initially began with creating a pipeline using XGBoost[6]. XGBoost is a well-known, open-source Python library that implements gradient boosting machine learning algorithms for classification and regression. Zephyr offers predefined model pipelines that provide access to these algorithms. Thus, once the label and feature engineering steps are concluded, the user moves on to select the model pipeline through Zephyr, and to evaluate it using

receiving operator curves, confusion matrices and evaluation metrics.

Figure 3-6 shows an XGBoost pipeline used in the case study (Chapter 5).

The pipeline first trains an XGBoost model, which produces the predicted $\hat{y}$. Zephyr ML pipelines automatically output various model evaluation metrics including F1 score, accuracy, and test/train rate. Additionally, the user can use a post-processing primitive (`find_threshold`) to get the threshold up to the desired true positive rate. Through this function, the user passes as input the desired true positive rate (or any other metric), and can obtain the threshold that achieves it. This threshold is a learned parameter that is then applied to model output, generating a variety of metrics. In our experience, this functionality is critical for users since there are trade-offs between different metrics, and users often wish to control at least one of them.



(a) Pipeline graph

```python
from zephyr_ml import Zephyr

zephyr = Zephyr('xgb')

zephyr.fit(X_train, y_train)

y_hat = zephyr.predict(X_test)
```

(b) Python SDK

Figure 3-6: XGBoost example for (a) a graph representation of a pipeline (b) a code snippet for training the pipeline using the Zephyr API, then applying inference on the test set.

Chapter 4

Subject Matter Expert Inputs

A key feature of the Zephyr framework is that it enables users, in this case Subject Matter Experts (SMEs), to incorporate domain expertise into each step of the process. SMEs have a deep understanding of turbines and the associated data, the problems that befall them and the contexts in which they occur, and key insight into relevant and important features. They may also have access to additional data that may be relevant to the problem and influence the results. This Chapter highlights the more refined controls within the Zephyr framework, and how SMEs can adapt it to better fit their specific problems and own needs.

4.1 Data Input

Zephyr is designed to work with signal and operations data that is relatively standardized across the field, as described in Section 3.3. However, the *entityset* structure allows for significant flexibility in the data ingestion step, allowing SMEs to leverage their knowledge to ensure compatibility between their data and the framework and add or remove data.

First, Zephyr uses default names for key columns. For example, `COD_ELEMENT` is used for the unique identifier for each turbine in the `turbines` entity, and `DAT_START` is used as the time index for the `alarms` entity. This is done for convenience since most data sources follow similar naming conventions. However, the column names

associated with these key columns may differ between different organizations or data sources, so Zephyr offers the ability to change the default column mapping to match users' actual data. Once a new mapping is defined, it may be used as an additional argument to each step in the Zephyr framework to ensure compatibility throughout the process.

Second, Zephyr can easily ingest new data for existing entities by updating the underlying dataframe. For example, SMEs may want to include additional data as it becomes available or expand a prediction problem to cover more turbines and their corresponding data. This data can simply be appended to the *entity*'s data structure to be added to the *entityset*.

Finally, Zephyr can adapt to entirely new sources of data by adding new *entities*. SMEs may have access to additional data beyond the signal and operation data that Zephyr uses, and wish to use it in the framework. A new entity can be added to an *entityset* using the `EntitySet.add_dataframe` method, where *entity* metadata including specifying key columns and column typing can optionally be given. A new relationship mapping the new entity to an existing entity in the *entityset* must also be given through the `EntitySet.add_relationship` method. Once the new data has been placed in an entity in the relational *entityset* structure, it can be used in all future steps.

4.2 Label Engineering

The size of the `prediction window`, `gap`, and `lead` parameters all offer fine-grained control over the data labeling process. SMEs can take advantage of this to tune the structure of a prediction problem or control the number of labels generated. For instance, the length of the `lead` parameter determines the number of days ahead of a which a component failure prediction is provided, in order to give reasonable warning of impending failure. Thus, when a failure is predicted, the time frame for a response is known and can be mapped into operating and maintenance processes. Furthermore, if multiple models with different `lead` parameters are trained, they can

be put in operation in parallel to give multiple detection points (warnings and alarms etc.). The `prediction window` and `gap` parameters are generally highly domain and problem specific, as they determine how the data is segmented to search for examples directly influencing cutoff times and associated labels. SMEs can fine tune these parameters to ensure labeling functions are optimized in how they search for labels to fit real-world expectations.

In the **BP** case study, for example, three `lead` times were used – 0, 28 and 42 days – while in the **CF** case study experts used 0, 7 and 10 days. In early iterations, the `lead` parameter was minimized to zero days to ensure the prediction problem was viable simply for failure detection during late failure progression. Importantly, generating models for new `lead` times only requires applying a new `lead` to the cutoff times and re-running the feature engineering and model generation steps. The isolated step-by-step nature of the Zephyr framework makes rapid iteration possible since modifications to one step do not require side-effect modifications in future steps.

While the `prediction window`, the `gap` and the `lead` all influence how data labels are produced, the labeling function impacts what the labels represent directly. In Zephyr, three predefined labeling functions are available: one that checks if a converter was replaced, another that checks for the presence of a brake pad failure, and a third that calculates the total power loss over a data segment. Additionally, as described in Section 3.4, Zephyr provides the ability to easily create custom labeling functions. With a few lines of code, SMEs can create new labeling functions that can be used directly without additional intensive development. This flexibility means that Zephyr can easily be expanded by SMEs to allow them to incorporate their knowledge and explore new prediction problems.

4.3 Feature Engineering

4.3.1 Signal Processing

SMEs can fully leverage their knowledge of important signals and variables to guide feature engineering. Given raw signal data, SMEs have a deep understanding of what transformations and aggregations might lead to relevant features. `SigPro` easily enables custom signal processing pipelines that can perform commonly applied signal transformations and aggregations. These signal processing pipelines are further tailored by the SMEs to fit the problem through the selection of hyperparameters such as specifying relevant signal bands or side bands.

4.3.2 Feature Generation

SMEs can also improve automated feature engineering by selecting aggregation or transformation functions (called primitives in our library) and using seed features. SMEs have the domain knowledge required to know which operations may lead to useful features when applied to their data. By selecting which primitives to use during feature generation, SMEs can apply their domain knowledge to ensure relevant features are being generated and help limit the number of irrelevant features that may get generated.

Seed features allow SMEs to describe custom features (known symptoms or root causes, for example) in order to generate additional important features beyond those that are automatically generated. SMEs use their knowledge of past failures that have been subjected to root cause analysis (RCA), failures included in the FMECA and symptoms analysis, and other failure analyses to find contributing factors and symptoms. They then transform these factors into seed features that are used to enhance the feature engineering step.

Finally, individual columns or entire *entities* can be ignored during the generation process to avoid creating features that are known to be irrelevant to the problem or that may contribute to data leakage. For example, in the converter case study, the

SMEs dropped the `notifications` table during the feature engineering step because it did not contain useful information.

4.4 Model Evaluation

To evaluate a model, an SME has a number of options available. They can select the desired true positive rate to find an appropriate threshold. They can apply an F-beta score with a beta set to 0.5, 1, and 2 to achieve a more deterministic balance (precision orientated, balanced precision-recall or recall orientated), so that the outcomes are more certain and can be matched to the criticality of failure and appropriate follow-up response. They can adopt other metrics such as ROC-AUC, or they can incorporate subject knowledge to determine the most salient metrics to use. Additionally, a sensitivity analysis is conducted on the F-beta scores with respect to the class threshold to inform the SME of the optimal class threshold when making the model operational for inferences. These can then be fed back to the standard confusion matrix for recalculation at the optimal threshold, as well as the other evaluations metrics, which are calculated based on a default 0.5 class threshold.

Chapter 5

Evaluation

To evaluate our framework, we worked with a group of SMEs with experience in machine learning and data science, as well as a group of data scientists and machine learning experts without wind turbine expertise (Non SMEs), to use Zephyr to create models for both case studies independently. Our goal is to emphasize the importance of integrating domain knowledge into model development and highlight that users can effectively utilize our framework even without significant machine learning expertise. In this Chapter, we highlight the differences between the approaches taken by the Non SMEs and the SMEs, and how the SMEs leveraged their knowledge to achieve better results.

5.1 Case Studies

5.1.1 Setup

The **B**rake **P**ad (**BP**) case study involves five *entities*: `turbines`, `notifications`, `stoppages`, `work orders`, and `SCADA signals`. The **C**onverter **F**ailure (**CF**) case study involves six *entities*: `turbines`, `alarms`, `notifications`, `stoppages`, `work orders`, and `PI signals`. Table 5.1 lists the data used for both case studies.

	BP		CF	
Table	#Rows	#Columns	#Rows	#Columns
Turbines	70	10	150	10
Alarms	-	-	6566850	10
Notifications	127	15	32515	15
Stoppages	16898	18	113654	18
Work Orders	5735	21	4090	20
SCADA	7910640	167	-	-
PI	-	-	20219255	35

Table 5.1: This table reports characteristics of the case study datasets, in particular the number of rows (#Rows), and the number of columns (#Columns).

5.1.2 Brake Pad Case Study

Data Loading

To fairly evaluate the resulting models, both the SMEs and Non SMEs used the same data. After loading in the data for the brake pad case study, presented in Table 5.1, both sets of users created the *entityset* through the Zephyr function `create_scada_entityset()`.

Labeling

In this case, both the SMEs and Non SMEs used the predefined function `brake_pad_presence` which was co-developed with SMEs. The Non SMEs did not change the predefined default parameters, since they do not have a deep knowledge of the data or the context around the problem. The labeling function uses the following features:

- The text description associated with the stoppage,
- The ID of the turbine experiencing the stoppage, and
- The end time of the stoppage, which serves as the time index.

The function checks whether the text description of a stoppage contains references to brake pads, indicating the brake pads were at fault for the stoppage. With their

search, Non SMEs produced nearly balanced labels: 37 True and 33 False. The SMEs produced 943 True labels and 1498 False ones, for a total of 2441 labels; i.e. an average of 34 labels for each turbine. The number of labels differed drastically because the Non SMEs relied on the default `window size` of all future data and used a matching `gap`, producing only one label for each turbine. The SMEs used a `gap` of 20 days, resulting in significantly more labeled examples. The `gap` is modified because it must account for an estimated time window that considers multiple work order replacements for the same component in the window as a single failure incident.

Additionally, both the Non SMEs and the SMEs tried applying different amounts of `lead` time, in order to test trade-offs between model accuracy and earlier detection of future failures. They developed models for `lead` times of 0 days, 28 days, and 42 days. These `leads` are chosen because brake pad failures are more promising of having a longer `lead` time and period of wear, where abnormality can be detected. Additionally, the brake pad issue is most problematic in offshore wind turbines, where access due to planning and weather conditions is restricted.

Feature Engineering

The third step is feature engineering. Both sets of users used Featuretools' Deep Feature Synthesis (DFS) [14] in order to automatically generate features to train a better model. Non SMEs automatically selected the most frequently occurring categorical values from the turbine table as interesting values and generated features specific to those values. Other parameters in the feature generation processes were left to default values, and the Non SMEs were able to generate 1035 features.

The SMEs leveraged their domain knowledge through seed features and careful primitive selection (Section 4.3. They created four seed features for the SCADA data:

- `WROT_Brk1HyTmp1_mean` > 80
- `WROT_Brk1HyTmp2_mean` > 80
- `WROT_Brk2HyTmp1_mean` > 80

– `WROT_Brk2HyTmp2_mean` > 80

These seed features create boolean variables for when the average temperature of the brake pad exceeds 80°, a metric that is known to be a potential indicator of an impending failure. They also limit aggregations to `count`, `sum`, `percent true`, and `max`. These operations tend to reveal trends that indicate the health and status of a turbine over time. The default transformation primitives were used.

The SMEs chose to ignore `notifications`, `alarms`, and `work orders` when automatically creating features, as the SMEs intuited that they do not contain data that is useful for indicating imminent brake pad failure. By inserting domain knowledge during the feature engineering step, the SMEs generated a total of 318 features. Importantly, while the SMEs generated fewer features, these features were more relevant to the problem than those created through the scattershot approach taken by the Non SMEs.

Modeling and Evaluation

Finally, both sets of users generated an XGBoost Classifier [6] based on a tree model with an approximate greedy algorithm and a uniform sampling method. The model performance suffers significantly without domain knowledge. Non SMEs obtained AUC = **0.518** with 0 `lead`, AUC = **0.514** with 28 `lead`, AUC = **0.550** with 42 `lead`, while the AUC of the SMEs are **0.999**, **0.998**, and **0.996** respectively.

One example that highlights the importance of domain knowledge is, with a `lead` time of 0, the most important feature for the model created by the SMEs was built off a seed feature, specifically `PERCENT_TRUE(SCADA.WROT_Brk1HyTmp2_mean>80)`. As `lead` times increase, the importance of this feature falls off, but still remains in the top 10. The SMEs also saw a benefit in easily extracting feature importance, as this allows them to validate their feature engineering assumptions, as well as to use the feature importance as feedback to other stakeholders in order to demystify ML methods and models and increase user confidence.

Additionally, while performance inevitably falls off as `lead` time increases, being able to tune `lead` times allows the SMEs to holistically evaluate each model and

consider the trade-offs between model accuracy and early detection. Whereas Non SMEs may only be able to evaluate a model using traditional metrics, the SMEs have a more nuanced understanding of when to tolerate drops in performance in exchange for real-world benefits like earlier detection.

5.1.3 Converter Failure Case Study

Data Loading

As in the previous case study, to fairly evaluate the resulting models, both the SMEs and Non SMEs used the same data. After loading in the data for the converter failure case study – that is, the tables `turbines`, `alarms`, `stoppages`, `work orders`, `notifications`, `PI data`, detailed in Section 2.3.2 – both sets of users created the *entityset* through the Zephyr function `create_pidata_entityset()`.

Labeling

Both SMEs and Non SMEs used the same predefined labeling function `converter_replacement_presence`. This labeling function was co-developed with SMEs, who identified relevant columns and logic for determining label value. The `converter_replacement_presence` labeling function uses the following columns by default:

- The SAP code for the notification,
- The short description of the notification,
- The ID of the turbine associated with the notification, and
- The time the malfunction started, which acts as the time index.

The function requires that the SAP code associated with the notification be equal to 36052411 – indicating that a replacement did happen – in order to obtain a label of `True`.

SMEs and Non SMEs used different parameters in the label search process. Non SMEs used the default parameters for label generation, i.e., `window size` equal to 10 days and `gap` equal to 1. The SMEs used the default `window size` but changed the `gap` to 10 days. As in the brake pad case study, this parameter is modified because SMEs want to consider multiple repairs as one failure. The SMEs preferred transparency in the parameters so that they could control the time intervals, which would be aggregated to shape meaningful features and better represent trends in assets over time. Both the Non SMEs and the SMEs tried three different `lead` times: 0 days, 7 days and 10 days. These `lead`s are shorter than the `lead`s applied in the brake pad case study because converter failures usually do not have a long period of deterioration before a failure occurs. Also, the converter failure case study refers to onshore wind turbines, where access is easier and significant `lead` time to handle impending failures is not as critical.

Both the Non SMEs and the SMEs obtained unbalanced labels with their parameters, ending up with 2238 False and only 74 True labels. Because failures in the dataset were rare and the same `prediction window` was used, the `gap` did not change the number of labels produced for the SMEs.

Feature Engineering

The Non SMEs automatically selected the most frequently occurring categorical values from the alarms table to generate features specific to those values. Other parameters were left as defaults, resulting in the generation of 47 features.

The SMEs were able to leverage their knowledge during the feature creation step through seed features and careful primitive selection (Section 4.3). Primitive selection allows SMEs to further focus the power of deep feature synthesis on the most illuminating mathematical relationships between entities in the data.

One seed feature was created for the PI data: `WGEN.W_max` > 2000. This seed feature creates a Boolean variable indicating whether the maximum power generated over the interval exceeds some threshold. This seed feature was included as an explicit feature to indicate that the asset is operating at or close to its limit. Due to the lack of

electrical measurements, additional seed features relating to reactive power and grid frequency voltage were considered and implemented in later runs, without significant gains in performance.

The SMEs limited aggregations to `count`, `sum`, `percent true`, `min`, and `max`. These operations tend to reveal trends that indicate the health and status of the turbine over time. The `num words` transform primitive was also used, which determines the number of words for each row in a text column.

The SMEs also ignored the `notifications` entity when automatically creating features. This was done to limit the number of features generated, because they knew this data was unlikely to contain features useful for indicating imminent converter failure. The SMEs generated 1241 features in total, significantly more than the 47 features the Non SMEs produced.

Modeling and Evaluation

Finally, both the Non SMEs and the SMEs concluded the analysis using an XGBoost Classifier [6] based on a tree model with an approximate greedy algorithm and a uniform sampling method. Both users were instructed to use the same method for a fair comparison. In this case study, both the Non SMEs' and SMEs' models had performed poorly, and leveraging the domain knowledge of the SMEs did not help lead to significant performance improvements. Non SMEs obtained $AUC = \mathbf{0.540}$ with 0 `lead`, $AUC = \mathbf{0.542}$ with 7 `lead`, $AUC = \mathbf{0.558}$ with 10 `lead`, while the AUC of the SMEs are **0.514**, **0.604**, and **0.641**.

5.1.4 Ablation Study

Table 5.2 compares the evaluation metrics obtained on the test set for the BP case study. Particularly, we illustrate the comparison between Non SMEs and SMEs by removing some of domain information provided by the SMEs. Because SMEs injected information through parameters at different levels of the pipeline, we computed the metrics by reverting each parameter to it's default value, one parameter at a time. For

example, reverting the `gap`, or not adding seed features. The `gap` parameter has the most significant impact on the results, although other parameters were responsible for slight improvements as well.

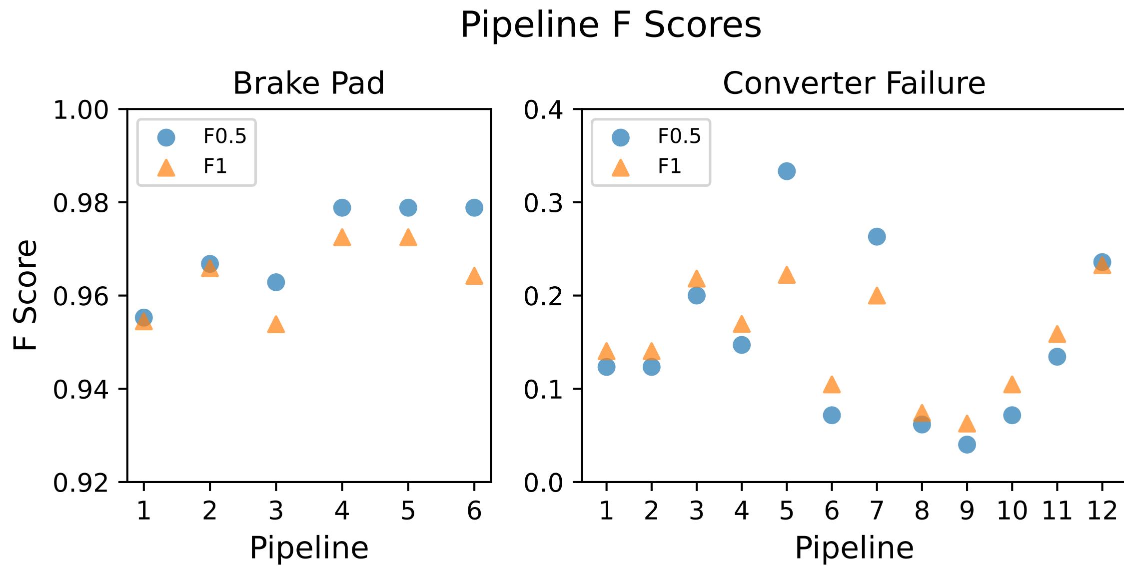


Figure 5-1: F1 Scores for the two case studies. Results show the pipelines crafted by SMEs and their respective F1 scores using Zephyr on their Azure platform.

5.1.5 Discussion

Figure 5-1 illustrates the different pipelines created by SMEs to solve the case studies. In **BP**, SMEs reached a satisfying F1 Score of 0.97 (F0.5 0.98) in their 4th and 5th iterations. However, in **CF**, SMEs investigated over 12 configurations to improve their model. While pipeline 5, 7, & 12 are the highest-scoring pipelines, at the time of writing SMEs are still manipulating the parameter settings such that they are able to train a reliable model. This is not an easy task, as many parameters (mainly in the early stages of the framework, e.g. the labeling phase) dictate the model performance. To overcome this challenge, Zephyr needs to connect its parameters to model performance.

5.2 Real-World Deployment

SMEs launched Zephyr on their Azure platform for all their team members to use. After preparing a compute cluster environment on Azure, they were able to install Zephyr through PyPI using `pip` installation and then design the results, files and graphs that need to be registered. Once that is accomplished, they are able to run the code and view the results of their experiments. Overall, the process took roughly a couple of weeks to complete. As part of their initial deployment, they launched both case studies on Azure. The pipelines in Figure 5-1 were all produced from their Azure platform. For each case study, they experimented over many trials, analyzing the model performances as they changed the data and certain parameters such as the `gap` and the `lead`.

5.2.1 Addressing Challenges

The wind energy industry still faces various challenges in turbine maintenance – particularly the three specific challenges the SMEs emphasized for us, as outlined in Section 2.3.3: the sheer amount of data to be monitored, the lack of context and labels in the raw data, and difficulty working with existing machine learning tools. Zephyr aims to solve these three main problems and to help SMEs develop predictive maintenance systems. Zephyr is an automated, scalable and flexible framework that allows SMEs to predict wind turbine component failures and to incorporate their knowledge into the machine learning process in order to improve predictive results. The data we used to predict brake pad presence or converter failure was composed of 5 and 6 tables respectively, some with more than 2M rows. Zephyr combines and analyzes vast amount of available signal data, addressing the problem with handling the significant quantity of data available for turbine maintenance.

The Zephyr pipeline begins with integration of data into *entityset*s, followed by label creation according to the specific labeling function. SMEs can define their own labeling functions according to the problem at hand, allowing SMEs to easily generate labels for any prediction problem from the data.

Last but not least, Zephyr integrates ML power with SME expertise to achieve the best possible predictions. SMEs with mechanical and electrical engineering backgrounds used the framework. They were able to inject their knowledge into Zephyr and use machine learning for predicting the failures in wind turbines. This eliminates the gap between SMEs and machine learning tools, making it easier for SMEs to use. Although Zephyr is very helpful to SMEs, many SMEs are not used to applying data-centric machine learning frameworks. Zephyr's specificity is advantageous here, because users must only learn a single, simple framework. While many of the tools Zephyr uses come with rich and flexible feature sets, many of these options may not be relevant for a given problem, and ensuring compatibility between these libraries can generate additional issues. Zephyr ensures compatibility with underlying libraries and reduces the learning cost for users.

	Brake Pad Case Study			Converter Failure Case Study		
Lead	0	28	42	0	7	10
Non SMEs	0.571	0.353	0.636	0.125	0.103	0.066
SMEs without changing the gap	0.476	0.688	0.688	**0.128**	**0.222**	**0.200**
SMEs without adding seed feature	0.968	0.932	0.930	0.070	0.200	0.092
SMEs without modifying the aggregation primitive	0.980	0.929	0.917	0.080	0.182	0.189
SMEs	**0.986**	**0.956**	**0.951**	0.127	**0.222**	**0.200**

Table 5.2: F1 scores using different lead times. Non SMEs stands for Non Subject Matter Experts, while SME stands for Subject Matter Experts.

Chapter 6

Conclusion

This thesis describes Zephyr, a data-centric framework for predictive maintenance of renewable energy assets. Zephyr allows users to aggregate wind turbine data, generate labels, and perform feature engineering, as well as model training and evaluation. Zephyr's most significant attributes are that it allows domain experts to input their knowledge at each step of the process, and that it easily enables iterative development over each of the steps.

In the case studies, we show that Subject Matter Experts (SMEs) obtained much better results than those obtained by Non SMEs. More specifically, SMEs improved the AUC score by 48% for the brake pad case study and 8.3% for the converter failure case study. Because they have deep knowledge and understanding of the data, SMEs are able to collaborate intelligently with Zephyr – inserting new relevant data, adding primitives and features, and changing label definitions and search parameters.

The SMEs also found significant value in Zephyr's iterative development structure and flexibility. They could easily evaluate outputs at each step of the pipeline, and intuit possible improvements in data labeling or feature engineering. This also allowed them more flexibility in evaluation – for example, allowing them to tweak the prediction engineering by applying different lead times, and to evaluate the resulting models both numerically and in the context of overall real-world utility.

To conclude, Zephyr is a flexible machine learning framework that allows subject matter experts to generate and execute pipelines for standard wind turbine data.

Zephyr enables subject matter experts to input their knowledge and easily iterate over parts of the analysis to obtain the best results.

6.1 Future Work

While there has been great progress in Zephyr as an end-to-end framework for predictive maintenance of wind turbines, there are future steps that can to be taken to improve useability and stability of the project.

6.1.1 Library Development

As an open-source library, maintenance and long-term stability of the project are important for the SMEs relying on the project. Dependency management, documentation, collaboration, and release management are all critical infrastructural issues faced by many open-source projects. We have made some strides in improving these pain points. For example, we include regular automatic testing of Zephyr on new releases of dependencies to catch dependency errors early. In addition, the documentation includes demo Jupyter notebooks to enable users to quickly start experimenting with the library. However, future work on improving the release process, making improvements to the documentation, and maintaining dependencies will ensure the stability required for real-world adoption and usage.

6.1.2 Improvement of Prediction Problems

Currently, Zephyr supports three prediction problems out of the box with predefined labeling functions. While Zephyr offers significant flexibility in creating custom labeling functions, additional commonly used labeling functions can be developed and added to reduce further reduce user workload.

Additionally, with the flexibility available to users through the various parameters available at each step of the process, optimizing a model's performance became a more nuanced problem. SMEs are keen to connect the logic of the parameters in Zephyr

to the model's general performance. In other words, what is the general guidance in choosing these parameters? Improving Zephyr's ability to guide parameter choice, especially for users with little to no experience with prediction engineering, is critical for continuing to bridge the gap between SMEs and model development.

Bibliography

[1] *Human First*, August 2020.

[2] What is sap?: Definition and meaning, 2022.

[3] Moez Ali. *PyCaret: An open source, low-code machine learning library in Python*, April 2020. PyCaret version 1.0.0.

[4] Sarah Alnegheimish, Najat Alrashed, Faisal Aleissa, Shahad Althobaiti, Dongyu Liu, Mansour Alsaleh, and Kalyan Veeramachaneni. Cardea: An open automated machine learning framework for electronic health records. In *2020 IEEE 7th International Conference on Data Science and Advanced Analytics (DSAA)*, pages 536–545. IEEE, 2020.

[5] Bindi Chen, Peter C Matthews, and Peter J Tavner. Wind turbine pitch faults prognosis using a-priori knowledge-based anfis. *Expert Systems with Applications*, 40(17):6863–6876, 2013.

[6] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 785–794, 2016.

[7] Katharina Fischer, Karoline Pelka, Sebastian Puls, Max-Hermann Poech, Axel Mertens, Arne Bartschat, Bernd Tegtmeier, Christian Broer, and Jan Wenske. Exploring the causes of power-converter failure in wind turbines based on comprehensive field-data and damage analysis. *Energies*, 12(4), 2019.

[8] Maryna Garan, Khaoula Tidriri, and Iaroslav Kovalenko. A data-centric machine learning methodology: Application on predictive maintenance of wind turbines. *Energies*, 15(3):826, 2022.

[9] Mari Cruz Garcia, Miguel A Sanz-Bobi, and Javier Del Pico. Simap: Intelligent system for predictive maintenance: Application to the health condition monitoring of a windturbine gearbox. *Computers in industry*, 57(6):552–568, 2006.

[10] Jyh-Yih Hsu, Yi-Fu Wang, Kuan-Cheng Lin, Mu-Yen Chen, and Jenneille Hwai-Yuan Hsu. Wind turbine fault diagnosis and predictive maintenance through statistical process control and machine learning. *Ieee Access*, 8:23427–23439, 2020.

[11] Guoqian Jiang, WeiPeng Fan, Wenyue Li, Lijin Wang, Qun He, Ping Xie, and Xiaoli Li. Deepfedwt: A federated deep learning framework for fault detection of wind turbines. *Measurement*, 199:111529, 2022.

[12] Xin Jin, Yiming Chen, Lei Wang, Huali Han, and Peng Chen. Failure prediction, monitoring and diagnosis methods for slewing bearings of large-scale wind turbine: A review. *Measurement*, 172:108855, 2021.

[13] James Max Kanter, Owen Gillespie, and Kalyan Veeramachaneni. Label, segment, featurize: a cross domain framework for prediction engineering. In *2016 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, pages 430–439. IEEE, 2016.

[14] James Max Kanter and Kalyan Veeramachaneni. Deep feature synthesis: Towards automating data science endeavors. In *2015 IEEE International Conference on Data Science and Advanced Analytics, DSAA 2015, Paris, France, October 19-21, 2015*, pages 1–10. IEEE, 2015.

[15] K[atherine] Wang. A machine learning framework for predictive maintenance of wind turbines, 2020.

[16] Kevin Leahy, Colm Gallagher, Peter O'Donovan, Ken Bruton, and Dominic TJ O'Sullivan. A robust prescriptive framework and performance metric for diagnosing and predicting wind turbine faults based on scada and alarms data with case study. *Energies*, 11(7):1738, 2018.

[17] Zepeng Liu and Long Zhang. A review of failure modes, condition monitoring and fault diagnosis methods for large-scale wind turbine bearings. *Measurement*, 149:107002, 2020.

[18] Leon Mishnaevsky Jr. Root causes and mechanisms of failure of wind turbine blades: Overview. *Materials*, 15(9):2959, 2022.

[19] Trinh Hoang Nguyen, Andreas Prinz, Trond Friisø, Rolf Nossum, and Ilya Tyapin. A framework for data integration of offshore wind farms. *Renewable energy*, 60:150–161, 2013.

[20] Neoklis Polyzotis and Matei Zaharia. What can data-centric ai learn from data and ml engineering? *arXiv preprint arXiv:2112.06439*, 2021.

[21] Matti Niclas Scheu, Lorena Tremps, Ursula Smolka, Athanasios Kolios, and Fergal Brennan. A systematic failure mode effects and criticality analysis for offshore wind turbine systems towards integrated condition based maintenance strategies. *Ocean Engineering*, 176:118–133, 2019.

[22] David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. Hidden technical debt in machine learning systems. *Advances in neural information processing systems*, 28, 2015.

[23] Onder Uluyol, Girija Parthasarathy, Wendy Foslien, and Kyusung Kim. Power curve analytic for wind turbine performance monitoring and prognostics. In *Annual Conference of the PHM Society*, volume 3, 2011.

[24] Wenxian Yang, Richard Court, and Jiesheng Jiang. Wind turbine condition monitoring by the approach of scada data analysis. *Renewable energy*, 53:365–376, 2013.