# A Machine Learning Framework for Predictive Maintenance of Wind Turbines

by

Katherine Wang

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

Masters of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2020

© Massachusetts Institute of Technology 2020. All rights reserved.

Author . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Department of Electrical Engineering and Computer Science
January 29, 2020

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

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

# A Machine Learning Framework for Predictive Maintenance of Wind Turbines

by

Katherine Wang

Submitted to the Department of Electrical Engineering and Computer Science
on January 29, 2020, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Electrical Engineering and Computer Science

## Abstract

Wind energy is one of the fastest growing energy sources in the world. However, the failure to detect the breakdown of turbine parts can be very costly. Wind energy companies have increasingly turned to machine learning to improve wind turbine reliability. Thus, the goal of this thesis is to create a flexible and extensible machine learning framework that enables wind energy experts to define and build models for the predictive maintenance of wind turbines. We contribute two libraries that provide experts with the necessary tools to solve prediction problems in the wind energy industry.

The first is `GPE`, which translates and uses the desired prediction problem to generate machine learning training examples from turbine operations data. The other library, `CMS-ML`, provides the architecture for building machine learning models using vibration data generated by turbine sensors within the Condition Monitoring System (CMS). With this architecture, we can easily create modular feature engineering and machine learning pipelines for the CMS signal data. Finally, we demonstrate the application of these two libraries on proprietary wind turbine data and analyze the effects of their parameters.

Thesis Supervisor: Kalyan Veeramachaneni
Title: Principal Research Scientist

# Acknowledgments

I would like to offer my special thanks to my advisor, Kalyan Veeramachaneni, for his guidance, mentorship, and support. The work presented in this thesis would not have been possible without him.

I would like to extend my thanks to the members of the Data to AI Lab of MIT LIDS for their support and resources. I am particularly grateful for the assistance given by Carles Sala. He provided invaluable guidance and feedback towards the design and implementation of the software packages developed as part of this work, in order to ensure the software quality is high and applicable in industry. In addition, I would like to thank Arash Akhgari for his help in creating the diagrams for this thesis and Cara Giaimo and Michaela Henry for their assistance in editing this work.

I would also like to thank the following projects which have provided the infrastructure and basis for this work: GitHub, PyPI, Tox, Pandas, Scikit Learn, Scipy, and PyWavelets. I would especially like to give thanks to the MLPrimitives, BTB, and MLBlocks libraries, as this thesis heavily utilizes these three modules.

In addition, this project would not have been possible without the support of Iberdrola. I would especially like to express my gratitude towards Timothy Fletcher and Robert Jones for their insights about the wind energy industry.

Finally, I wish to thank my parents, Ying and Donghui, and my friends for continuing to encourage and support me and my choices throughout my academic studies.

# Contents

**1 Introduction** **17**

1.1 Machine Learning for Wind Energy . . . . . . . . . . . . . . . . . . . . 18

1.2 Feature Engineering . . . . . . . . . . . . . . . . . . . . . . . . . . . 20

1.3 Pipeline Technologies . . . . . . . . . . . . . . . . . . . . . . . . . . 21

1.4 Main Contribution . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23

1.5 Thesis Organization . . . . . . . . . . . . . . . . . . . . . . . . . . . 24

**2 GreenGuard Prediction Engineering** **27**

2.1 Target Times . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

2.1.1 Target . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28

2.1.2 Target Entity . . . . . . . . . . . . . . . . . . . . . . . . . . 28

2.1.3 Cutoff Time . . . . . . . . . . . . . . . . . . . . . . . . . . . 28

2.2 Engineering Prediction Problems . . . . . . . . . . . . . . . . . . . . 29

2.3 Merging Event Data . . . . . . . . . . . . . . . . . . . . . . . . . . . 30

2.4 Prediction Engineering . . . . . . . . . . . . . . . . . . . . . . . . . . 33

2.4.1 Prediction Engineering Parameters . . . . . . . . . . . . . . . 33

2.4.2 Prediction Engineering for Wind Turbines . . . . . . . . . . . 34

2.5 Design Considerations . . . . . . . . . . . . . . . . . . . . . . . . . . 35

2.6 Library Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36

2.6.1 Project Structure . . . . . . . . . . . . . . . . . . . . . . . . 36

2.6.2 Data Input Format . . . . . . . . . . . . . . . . . . . . . . . 36

2.7 Data Pre-processing . . . . . . . . . . . . . . . . . . . . . . . . . . . 37

2.7.1 Parsing and Initial Processing . . . . . . . . . . . . . . . . . . 38

2.7.2 Linking Data Tables . . . 38
2.7.3 Processing Output . . . 39
2.8 Target Generation . . . 39
2.8.1 Labeling Functions . . . 40
2.8.2 Target Generation Function . . . 41
2.8.3 Prediction Engineering Parameters . . . 41
2.8.4 Integration with Compose . . . 42
2.9 GreenGuard . . . 42
2.9.1 GreenGuard Process . . . 42

**3 Machine Learning Framework for Condition Monitoring Data 45**
3.1 CMS Data . . . 45
3.2 Design Considerations . . . 47
3.3 Library Overview . . . 47
3.3.1 Project Structure . . . 48
3.3.2 Data Input . . . 48
3.3.3 Data Pre-processing . . . 49
3.3.4 Primitives . . . 49
3.3.5 Pipelines . . . 51
3.4 User Interaction . . . 53

**4 Analysis and Evaluation 55**
4.1 Data Overview and Analysis . . . 55
4.1.1 Data Overview . . . 55
4.1.2 Exploratory Analysis . . . 56
4.2 Initial Prediction Problems . . . 56
4.2.1 Yaw Failures . . . 57
4.2.2 Gearbox Problems . . . 57
4.3 GPE Parameters . . . 58
4.3.1 Processing Parameters . . . 58
4.3.2 Representation of Prediction Tasks . . . 58

4.3.3 Prediction Engineering Parameters . . . . . . . . . . . . . . . . 59
4.4 CMS-ML Usage . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
4.4.1 Data Preparation . . . . . . . . . . . . . . . . . . . . . . . . 62
4.4.2 Pipeline Details . . . . . . . . . . . . . . . . . . . . . . . . . 63
4.4.3 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
4.5 Concluding Remarks . . . . . . . . . . . . . . . . . . . . . . . . . . 64

**5 Conclusion** **67**
5.1 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
5.1.1 Improvement of Pipelines . . . . . . . . . . . . . . . . . . . . 67
5.1.2 GPE User Interface . . . . . . . . . . . . . . . . . . . . . . . 68
5.1.3 Direct Integration . . . . . . . . . . . . . . . . . . . . . . . . 69

**A Pipelines** **71**

# List of Figures

2-1 Generating a target from a data slice using a labeling function. The labeling function in this case is searching for whether or not the `yaw` keyword exists within the `comments` column. If this is the case for a data slice, then the labeling function returns a target of 1, representing true. Otherwise, the function returns 0, representing false. . . . . . . 30

2-2 Time overlap cases. (a) shows an example of the two full overlap cases, which both have an overlap percentage of 1.0. In (b), there is no overlap, which results in an overlap percentage of 0.0. Both cases in (c) have the same partial overlap percentage, because it is calculated in terms of the smallest interval. . . . . . . . . . . . . . . . . . . . . . . 32

2-3 Visualization of prediction engineering concepts. The training, forecast, and prediction windows all have a length of five days, and the labeling function generates a target of 1. . . . . . . . . . . . . . . . . . 34

2-4 Example usage of generalized labeling functions to create specific ones. We use `keyword_in_text` to create functions searching for the keywords *gear* and *pump* within designated columns. . . . . . . . . . . . 40

2-5 Relating the `readings` and `target times`. This diagram visualizes the process of extracting the training examples for each `target` based on the `cutoff time` and `turbine id`. The resulting data slice can then be passed to the machine learning model. . . . . . . . . . . . . . 43

3-1 Example FFT spectrum with the `y-values`, `offset`, and `delta (Δ)`. The x-axis contains the frequency values, while the y-axis contains the amplitude values. . . . . . . . . . . . . . . . . . . . . . . . . . . . 46

3-2 Metaprimitive usage. Metaprimitives can apply the primitive to either axis of the input matrix X, and can take in additional data and parameters. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52

3-3 Pipeline containing the metaprimitive which applies the primitive `cms_ml.statistics.crest_factor` to the data. . . . . . . . . . . . . 52

3-4 Feature engineering pipeline workflow. Within the dotted boundaries is the metaprimitive, which creates a chain with other metaprimitives if necessary. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53

3-5 An example of directly calling the function that extracts features from the CMS data. The aggregations applied are `mean` and `rms`. The CMS data is read in from the `./data` folder. Since an `output path` is specified, the feature engineering result is written to a file in the `./output` folder and nothing is returned. . . . . . . . . . . . . . . . 53

3-6 Command line interface with `mean` and `rms` as the aggregations. This command is equivalent to the Python function shown in Figure 3-5. . 54

3-7 An example of using the `MLBlocks` interface to make predictions with a `CMS-ML` pipeline. `cms-ml-pipeline` is the name of the pipeline. `X_train` is a data frame containing the `turbine ids` and `cutoff times` of the training data and `y_train` is a vector containing the associated targets. `X_test` contains the `turbine ids` and `cutoff times` of the testing data. `readings` is the processed CMS data. . . . . . . 54

4-1 Visualization of target times generated for the three prediction tasks. These targets represent the problem of predicting yaw-related maintenance events and are generated for a single turbine. . . . . . . . . . 61

4-2 The effects of the `min_data` parameter on the generated target times for a single turbine. If `min_data` is an integer, it represents the number of entries before starting the search process. `min_data` can also be a period of time. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61

4-3 The effects of the `gap` parameter on the generated target times for a single turbine. If the `gap` is an integer, it represents the number of entries to use a gap. `gap` can also be a period of time. . . . . . . . . . . 61

4-4 The effects of changing the `threshold` for converting the power loss output for a single turbine into a binary label. . . . . . . . . . . . . . . 62

4-5 Machine learning pipeline used to generate predictions from CMS data. 64

# List of Tables


2.1 Target times example with `turbine id` as the `target entity`. . . . . 29

2.2 The translation of prediction problems into labeling functions that can be used to generate targets. . . . . . . . . . . . . . . . . . . . . . . . . 30

2.3 Description of prediction engineering parameters. . . . . . . . . . . . 41

2.4 GreenGuard data input file format. . . . . . . . . . . . . . . . . . . . . 43

3.1 Descriptions of aggregation primitives. . . . . . . . . . . . . . . . . . . 50

3.2 Descriptions of transform primitives. . . . . . . . . . . . . . . . . . . . 50

4.1 Exploratory statistics of the data sources. . . . . . . . . . . . . . . . . 56

4.2 Effects of the `GPE` pre-processing parameters on the % of stoppages that correspond to a work order and notification. . . . . . . . . . . . 59

4.3 Problem statements for prediction tasks given by wind subject matter experts and their translation into corresponding parameters for training window, forecast window, prediction window. . . . . . . . . . . . 60

4.4 Prediction task target time statistics. . . . . . . . . . . . . . . . . . . . 60

4.5 Statistics on the feature engineering output for various parameters. . 65

4.6 `CMS-ML` prediction results. These metrics are not representative of what can be achieved once we refine the targets and optimize the machine learning pipeline. This is purely meant to demonstrate how the software works. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65

# Chapter 1

# Introduction

As machine learning has become more ubiquitous, it is now applied to an increasing number of domain-specific problems that were traditionally handled solely by experts [23]. However, models built for problems defined by machine learning engineers are not necessarily very useful, as the ML engineers can lack domain knowledge and not know what is specifically important in the domain. These systems may not offer the flexibility to suit the changing prediction needs of subject matter experts. Beyond identifying important problems, experts provide insights that can improve how the data is represented for machine learning. In addition, many machine learning frameworks do not focus on large-scale industrial deployment as the end goal, failing to provide a solution that is easily extensible to fit with existing systems and system formats. Through interactions with industry experts, we have found that these issues exist within the wind energy domain. Thus, a primary contribution of this thesis is providing a flexible machine learning framework towards the goal of enabling wind energy subject matter experts to easily process data and construct models for their desired prediction problems[1].

Wind energy has become one of the fastest growing sources of energy in the world due to its availability [10, 26]. However, maintenance of wind turbine parts can be costly and difficult, especially if the parts are difficult to access [1]. Thus, machine

[1]This project is under active development within the Data to AI Lab. While the thesis presents the first version of the software, much of the libraries, abstractions, and methods presented are in active development.

learning for the predictive maintenance of turbines is very important. Machine learning for wind energy is also of interest due to its challenging nature, in part due to the significant size of its signal data and the sparsity and occasionally difficult extraction of specific maintenance events.

This project has been carried out in partnership with a wind energy company which has provided both key insights and a significant amount of data related to their turbines, including signals and turbine operations data over the last few years. The industry experts have presented important information regarding the data and active feedback on the framework and models built using it.

Using the inputs from the subject matter experts (SME), we contribute the following components. The first is developing a framework to streamline the process of converting an outcome that a SME is interested in and the operations data into desired machine learning prediction problems. We also focus on the data from one of the sources of provided signals, called the Condition Monitoring System (CMS). CMS is further introduced in this chapter. We provide a modular framework for creating representations and building machine learning models for the raw CMS data.

This chapter introduces the motivation and context for a machine learning framework for use by wind energy experts, especially for the predictive maintenance of wind turbine parts. In this chapter, we provide an overview of previous work done in applying machine learning to wind energy systems. We also present an overview of the main concepts and technologies utilized in this work. This includes feature engineering, prediction engineering, and the pipeline technologies used to help build the framework. Finally, we discuss the main contributions of this thesis, along with its organizational details. This thesis is part of an ongoing project and reflects the current state of the associated libraries. The libraries will evolve as the project continues.

## 1.1 Machine Learning for Wind Energy

For wind energy companies, remote monitoring and early detection of events is extremely important due to the high cost of corrective maintenance and the impact of

outages. This is important because generators and other parts may be difficult to access immediately, especially for offshore wind farms [1]. The space has also become increasingly popular within the machine learning community, especially as wind energy becomes one of the fastest growing power sources [10]. It has a unique set of challenges due to the stringent performance and reliability requirements and the large size of the data [9].

Currently, many wind energy companies employ condition monitoring systems (CMS) to actively monitor for emerging problems, often through spectral analysis techniques [1, 28]. CMS systems typically consist of vibration-based sensors on turbine parts, especially expensive and essential parts like drive trains and blade components [28]. Vibration changes can potentially be indicative of a fault. When rotating parts suffer damages, it can often lead to significant vibrations, and different defects can generate different vibration patterns at specific frequencies [7]. Typical analysis methods span both the time and frequency domain, ranging from statistics to advanced spectral analysis techniques [7]. There is a lot of potential for extracting more insights from the CMS data based on the processing and analysis techniques that are chosen. Thus, due to its unique data source and the importance of its representation, the CMS data is a large focus of this thesis.

Along with CMS sensors, data is also collected through the large number of sensors within the supervisory control and data acquisition (SCADA) system. The SCADA system keeps track of both signals and alarms, usually at 10 minute intervals [28]. If the dataset is thorough, the dataset will include 10-minute averages, standard deviations, minimums, and maximums [7]. SCADA data includes, among others, data on external conditions and operational values from a variety of positions. These systems are set up so that when certain parameters exceed a threshold, the turbine stops [24].

Previous work related to predicting wind energy events are generally focused on the susceptibility-to-failure of certain turbines. Most works pursue one of three different approaches: turbine and sub-component models, signal processing models based on CMS data, and models based on the large amount of data provided through

SCADA systems [21]. Analyses and models using SCADA data are more common since the system is more widespread; virtually all modern turbines have the SCADA system installed [17]. For this project, access to both CMS and SCADA data is provided, allowing for the proposal of a wider range of models and a potentially better result, especially for the lesser explored CMS data.

Many models have used SCADA data to make short-term predictions of turbine stoppages and outages, generally through anomaly detection, classification methods and other machine learning techniques [4, 17, 24]. Other works, recognizing the impact that stoppage duration has on power generation, proposed a model for predicting stoppage duration, while others have created methods for failure detection during extreme events such as weather [5, 12]. But, one such group of authors, who were using neural networks for outage prediction, pointed out that a weakness of their system was that it was never deployed and tested, and they had no domain knowledge into how applicable their system would actually be [24].

It is clear that there are a large range of problem types that subject matter experts may want to predict or model, but most works do not engage experts to gain insights or determine what is useful, nor do they offer the flexibility for multiple problem types. There are works that have involved experts, but many do not have the flexibility or interactability that we propose in this thesis. For example, R.L. Hu, who has done much research in the wind energy space, involves knowledge garnered from subject matter experts to help manually create artificial features [10] or select for useful features [11].

## 1.2 Feature Engineering

This work applies the concepts of feature engineering to time series signal data in order to improve the performance of machine learning models. Specifically, feature engineering is the task of altering the representation of raw data to better fit a predictive modeling algorithm [15]. Feature engineering can help create better representations of data with incompatibilities, data with missing data values and data across multi-

ple databases [18]. Feature engineering can also help create smaller representations of very large datasets, a property that we utilize in this work.

The performance of machine learning models are fairly dependent on feature representation, resulting in a lot of time spent on data transformations [2]. The derivation of successful feature engineering representations typically involves domain knowledge and a labor-intensive trial-and-error method [15]. The set of representations shown in this work has been developed with significant subject matter expert input and builds upon previous work done on feature extraction on vibration signal data.

## 1.3 Pipeline Technologies

This project builds on top of work done on machine learning back-end automation and abstractions designed by members of the MIT Data to AI (DAI) Group. Specifically, this work builds on `MLBlocks`[2], `MLPrimitives`[3], and Bayesian Tuning and Bandits (`BTB`[4]). To understand the usage of the architecture of these three libraries, we introduce a few concepts:

- A **primitive** is a fundamental data processing block that performs a single, well-defined computation on the data and produces an output. A primitive has clearly specified inputs and outputs and, in some cases, hyperparameters that control how it works. There are two types of primitives:
  - The simplest type of primitive only has a data processing stage. Within this stage, called the *produce* stage, the primitive takes in input data and produces an output.
  - The other type of primitive also has a *fit* stage, preceding the *produce* one, where the primitive learns parameters from the data. These will be used later on to produce the output.

[2]https://hdi-project.github.io/MLBlocks/
[3]https://hdi-project.github.io/MLPrimitives/
[4]https://hdi-project.github.io/BTB/

- One or more primitives can be run sequentially to form a **pipeline**[5], where the output of one primitive is used as the input to the next. Pipelines have *fit* and *predict* stages. Within the *fit* stage, primitives are sequentially fitted and immediately used to produce transformed data used by the next primitive. In the *predict* stage, the primitives are only used to produce transformed data.

`MLBlocks`, is a Python-based open-source library, that provides a simple framework for creating machine learning pipelines out of primitives from a huge ecosystem of Python data science libraries [22]. It provides a JSON-based language to annotate pipelines and primitives, along with a run time to execute them. For primitives, the annotation specifies the inputs and outputs, as well as the hyperparameters and how to tune them. The pipeline annotation includes the sequence of primitives, their initialization parameters, and information about how to connect them, if needed. As the hyperparameters belong to the primitives, `MLBlocks` also extracts the hyperparameters from the primitive specifications into a machine-readable format so the pipeline can be tuned as a whole.

`MLPrimitives` is the open-source library that contains primitive and pipeline annotations, along with the code for custom primitives. The primitives include both custom and third-party methods from data science and machine learning libraries such as `Scikit Learn`, `Pandas`, and `Keras`. The primitive annotation also contains the hyperparameters that can be used, along with a carefully-selected ideal optimization range. `MLPrimitives` contains a selection of pipelines using these primitives. These pipelines are ready-to-use to solve a number of pre-defined machine learning problems.

To tune the hyperparameters extracted by `MLBlocks`, we use `BTB`, an open-sourced library which provides a framework for hyperparameter tuning, and pipeline selection [22] [8]. `MLPrimitives`, `MLBlocks` and `BTB` serve as the basis for both the `GreenGuard` and `CMS-ML` systems described in Chapters 2 and 3 respectively.

[5] In some contexts, a pipeline can be referred to as a *template* as it can be used to build similar pipelines.

## 1.4 Main Contribution

This work aims to empower subject matter experts by providing them the necessary end-to-end architecture and abstractions to construct models for relevant problems. The main contribution towards this is the development of two distinct libraries which contribute to the machine learning process for predictive maintenance on wind turbines.

The first library contributed by this work is a prediction engineering library called GreenGuard Prediction Engineering or `GPE`. `GPE` provides processing and linking methods for related operations data, and applies prediction engineering methods on the pre-processed data. `GPE` can ingest a variety of formats and operations data types. `GPE` allows users to easily define new prediction problems to generate data for, giving subject matter experts the flexibility to define problems as needs arise. `GPE` supplies the inputs for another library called `GreenGuard`, which together provide a complete machine learning solution.

The second library is a machine learning framework for the CMS data called `CMS-ML`. As powerful training data can be derived from CMS signal data, the goal is to provide users with a way to easily use and transform this data. We present a pipeline system with a common interface where different feature representations can be easily interchanged or even combined. New feature representations can also be easily added to allow users who may be more familiar with the input signal data to utilize their insight in the feature engineering process.

By extending the pipelines used for feature engineering to include machine learning primitives, we are able to build end-to-end, modular machine learning models. We are also able to incorporate hyperparameter tuning. Finally, other than providing an easy-to-use infrastructure, we also provide tested, ready-to-use pipelines that we have built for modeling predictive maintenance problems.

Finally, we also provide an exploratory analysis of the data sources, key wind machine learning insights based on interactions with industry experts, and the results from applying the libraries.

To summarize, with this thesis the following was achieved:

1. The creation of a prediction engineering library specifically for operations data (`GPE`).

2. The demonstration of the effects of `GPE` parameters on inputs for machine learning.

3. The creation of a feature engineering and machine learning library for wind turbine CMS signal data (`CMS-ML`).

4. The development of feature engineering and machine learning pipelines in `CMS-ML`.

5. Insights towards predictive maintenance machine learning from wind energy experts.

Overall, these contributions provide the infrastructure for machine learning for wind turbine predictive maintenance. They have been developed with constant feedback from our industry partners and are instrumental in progressing towards a deployable system for large wind energy companies to use. The work completed not only includes some ready-made examples, but also allows for future solutions to be easily built.

## 1.5 Thesis Organization

The structure of this thesis is as follows:

Chapter 2 and 3 introduce the design and implementation of the two main components of this work: the `GPE` and `CMS-ML` libraries respectively. `GPE` is a prediction engineering library for wind operations events, while `CMS-ML` is a machine learning library for CMS data. Within the `GPE` chapter, we also introduce `GreenGuard`, the last component of the machine learning architecture for turbine predictive maintenance. All have been designed to fit the needs of wind energy experts based on their extensive feedback.

Chapter 4 shows the analysis of the data and the evaluation of the software. The effects of user-inputted parameters are explored. Various machine learning pipelines, built using the components of this work, were attempted in order to solve a series of initial predictions problems. Chapter 5 summarizes the contributions made in the thesis, along with future goals for the project.

# Chapter 2

# GreenGuard Prediction Engineering

GreenGuard Prediction Engineering, or `GPE`, is a library that uses operations data from wind farms and finds *past occurrences of events* in order to generate training examples for machine learning. `GPE` automates and provides abstractions for the process of representing a prediction problem given raw data. It also offers flexibility and ease-of-use with regards to file formatting, the designation of parameters, and the definition of problems. In this chapter, we introduce the `GPE` library and relevant concepts, discuss the design principles, and delve into the implementation details of the data processing and functionalities that generate training examples.

## 2.1 Target Times

In this section, we introduce the concept of *target times*. Target times are a set of tuples that consist of three components: `target`, `target entity`, and `cutoff time`. Target times are important for machine learning because they tell the algorithm when in the past an event or a outcome that we are interested in predicting happened, so it can learn from those experiences. We now briefly describe each of the components.

### 2.1.1 Target

The `target`[1] is the value that we want to predict, and is representative of the problem we are aiming to solve. In this thesis, the target is either a binary or numerical value. An example would be a target denoting whether or not a system failure occurred. The target would be true if there was a system failure, and false otherwise.

### 2.1.2 Target Entity

The target entity is the real-world object on which we make predictions. In most cases, this means that the data used to produce a specific target is only associated with a singular instance of the target entity. In the specific case of this thesis, our target entity is a turbine, which has an associated `unique id` we refer to as the `turbine id`. This means we are making predictions for the turbine. A turbine also has signal readings from a multitude of sensors. We chose turbines instead of something like the signals to be able to use as many signals in the readings as needed.

### 2.1.3 Cutoff Time

Finally, the cutoff time is a datetime variable that is the point in time after which no data can be considered for a prediction. We use the cutoff time to help filter for signal readings that we can use as training data for predicting a certain target. The cutoff time can be important in the representation of prediction tasks. For example, if we wanted to make a prediction now for an event one week in the future, the cutoff time ensures that no data beyond the current point can be accessed for predicting the target.

An example of a set of target times is shown in Table 2.1.

[1] `target` and `label` are used interchangeably.

| turbine id | cutoff time | target |
|---|---|---|
| T001 | 2019-07-31 | 1 |
| T002 | 2019-08-15 | 0 |

Table 2.1: Target times example with `turbine id` as the `target entity`.

## 2.2 Engineering Prediction Problems

To generate the `target times` data frame, we need not only past data, but a process for representing the prediction problem that translates the past data into targets. We want to be able to functionally capture the definition of the outcome that subject matter experts are interested in predicting. However, setting the outcome and finding these past occurrences can be difficult. Though some outcomes can be easily computed by searching through the fields of the event data, some are difficult to compute and must be verified with a subject matter expert.

For those outcomes that are readily computable, we are able to capture the meaning of issues experts want to predict within a function called **labeling function**. A labeling function is a function applied to the data to calculate targets. The labeling function takes in a window of data that may or may not contain a past occurrence in order to generate a target. We call this specific window the **prediction window**, while a segment of data is generally referred to as a **data slice**. An example of applying of a labeling function to a data slice is shown in Figure 2-1.

We give an example where a subject matter expert wants to predict whether a turbine is going to stop due to issues with the yaw motor. This can then be translated into a labeling function that sets the target as true if the `yaw` keyword is in the `comments` column of a data slice containing a list of stoppage events, and false otherwise. We have represented the yaw stoppage with the existence of `yaw` in `comments`. Other examples of prediction problems and their associated labeling functions are shown in Table 2.2.

Once we have this method representing the problem that is to be predicted, we then need to search through the data for the appropriate data slices that we will apply these labeling functions to. We call the streamlined search process **prediction**

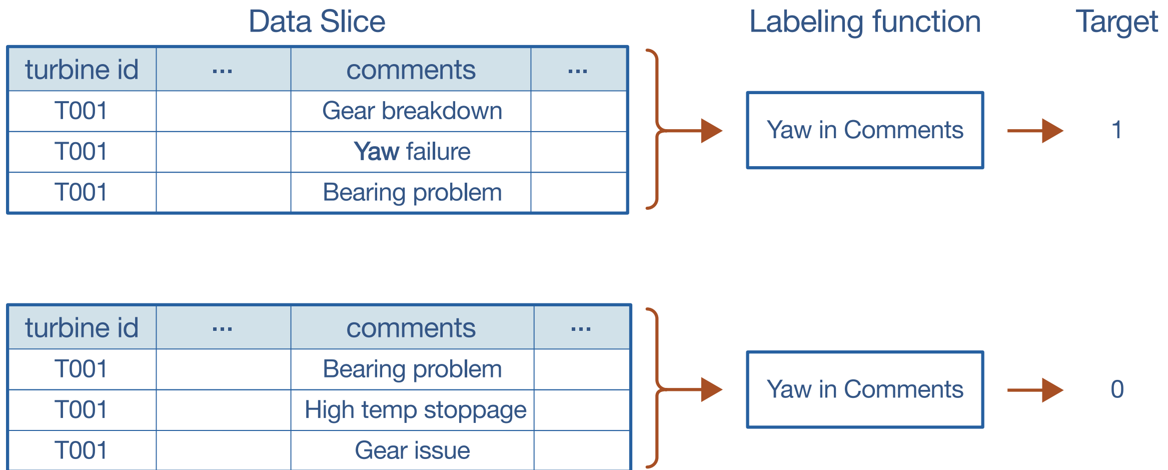



Figure 2-1: Generating a target from a data slice using a labeling function. The labeling function in this case is searching for whether or not the `yaw` keyword exists within the `comments` column. If this is the case for a data slice, then the labeling function returns a target of 1, representing true. Otherwise, the function returns 0, representing false.

| Prediction Problem | Labeling Function |
|---:|---|
| Predict if a yaw motor going to require repair? | Check for the existence of `yaw` & `motor` & `repair` in text fields |
| Predict if a gearbox bearing going to require inspection? | Check for the existence of `gearbox` & `bearing` & `inspect` or `investigate` in text fields |

Table 2.2: The translation of prediction problems into labeling functions that can be used to generate targets.

**engineering**, which is further detailed in section 2.4. Before prediction engineering can be applied, we need to be able to combine the disparate files to fully utilize the past data, a process which we introduce in the next section.

## 2.3 Merging Event Data

Ideally, any event-related data would be presented as one unified dataset, especially since our prediction engineering process currently expects a single table of events as an input[2]. However, more often than not, this is not possible. The 'parent' source of

[2]In future revisions, we can expect to accept multiple tables.

past occurrences and related 'child' data may be stored and recorded across different databases or tables, especially if the data can be generated from many independent sources.

The data we use for this thesis, for example, is sourced from independent processes. The 'parent' source of data is turbine stoppages, which are often automatically generated, while other 'child' sources of event data, such as work orders, are generated by technicians in the field and entered into a separate system. Thus, in order to fully utilize all of the data sources for producing target times, the data must be merged.

If two sets of data have the same primary key, the merging process is simple. However, if they have no relationship explicitly defined within the data, we try to infer the implicit relationship between entries of the two sets. One way to find this linkage is by calculating the time overlap percentage between the timestamps of entries of different tables and checking whether it is above a defined threshold. In our specific case, the 'parent' and 'child' table each have two timestamps, `start` and `end`. For stoppages, the `start` and `end` denote the period when the turbine was stopped, while for work orders the `start` and `end` indicate when the work on the turbine was conducted.

The time overlap percentage is calculated as follows, where $s_1$, $e_1$, $s_2$, $e_2$ mark the endpoints of the two intervals (interchangeable):

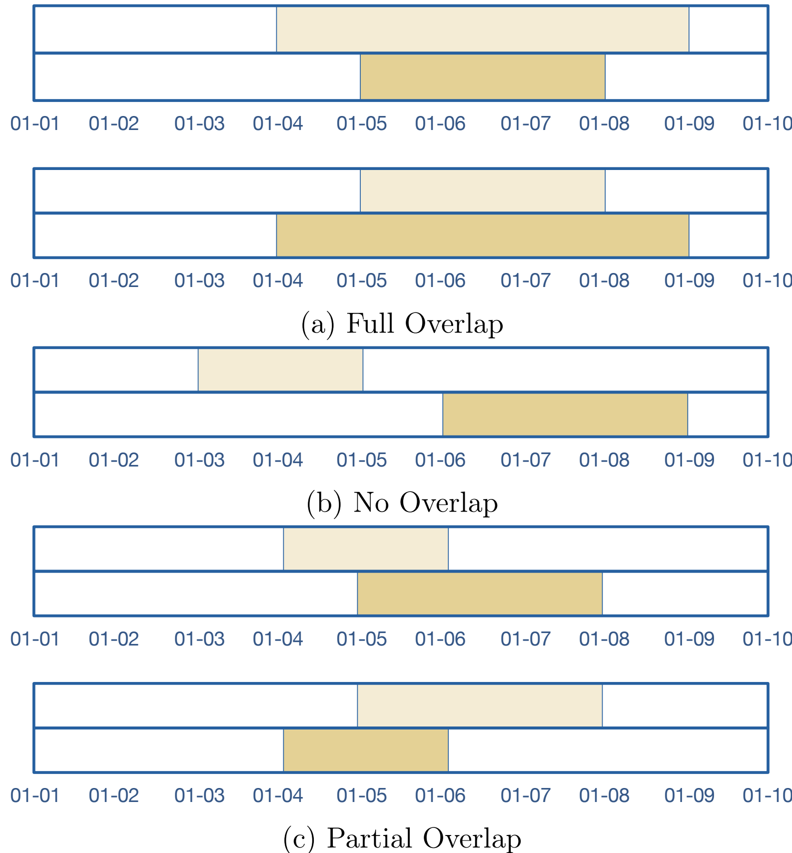
$$\max(0, \frac{\min(e_1, e_2) - \max(s_1, s_2)}{\min(e_2 - s_2, e_1 - s_1)})$$

This formula was derived to cover the three cases of interval overlap situations demonstrated in Figure 2-2. The first is when one interval entirely covers another interval. Regardless of which interval is enclosed, the overlap should be **1.0**. If there is no overlap at all, we want to ensure the percentage is exactly **0.0**. Finally, if the time intervals of two entries partially overlap, overlap percentage is calculated in terms of the smallest interval. This is because we may have, on average, much longer time ranges for one type of data compared to another. If almost all of one entry is within the time frame of another, then we should consider that a high amount of overlap.

Examples of relevant cases are shown in Figure 2-2.

To give an example of how the time overlap percentage is used to link data entries, we consider a case where our minimum overlap threshold is 0.5. If an entry from the 'parent' table has a calculated overlap of 0.8 with an entry from the 'child' table, these will be merged to form a single entry.

We may run into the problem that even with a time-based linkage, the relationship between the entries in the 'parent' and 'child' can be ambiguous. This is especially the case if we find time linkages to multiple 'parent' entries for a 'child.' To combat this, we can filter out linkages using information present in the text fields of the entries. While our library covers this case, it is beyond the scope of description here.




(a) Full Overlap

(b) No Overlap

(c) Partial Overlap

Figure 2-2: Time overlap cases. (a) shows an example of the two full overlap cases, which both have an overlap percentage of 1.0. In (b), there is no overlap, which results in an overlap percentage of 0.0. Both cases in (c) have the same partial overlap percentage, because it is calculated in terms of the smallest interval.

## 2.4 Prediction Engineering

Prediction engineering describes the conversion of timestamped data into feature vectors and targets [14]. In this thesis, prediction engineering is used in particular to represent the process of extracting labeled target times from related event files. The process borrows heavily from Kanter et al. [14], who define and outline the prediction engineering process in their work. They created a framework for prediction engineering consisting of 3 main actions: *label*, *segment*, and *featurize* [14]. Within the label step, the method for extracting targets from a slice of data is defined, and the data is traversed to search for targets. Each `target` is associated with a `cutoff time`, which marks the start of the data slice's window. The segment step generates data slices based on a number of prediction engineering parameters. The parameters are described in sections 2.4.1 and 2.8.3, as well as in [14]. Feature engineering is then applied to the data slice prior to the cutoff time to produce a feature vector for machine learning. The Label-Segment-Featurize model forms a basis for the prediction engineering methods used to develop the machine learning framework in this thesis.

### 2.4.1 Prediction Engineering Parameters

- The **training window** is the amount of past data that is used for the training a model that predicts the future after the cutoff time.

- The **forecast window** is the window between the end of the training window and the start of the prediction window. This window represents how far ahead to predict. For example, if we wanted to predict an event a week in advance, the forecast window would be one week. The length of the forecast window is also called the **lead**.

- The **prediction window** is the window after the forecast window. It is the window of data that is used to calculate the target. More specifically, the data in the prediction window is fed into the labeling function to generate a target.

A time series example using the prediction engineering terms defined above is shown in Figure 2-3.

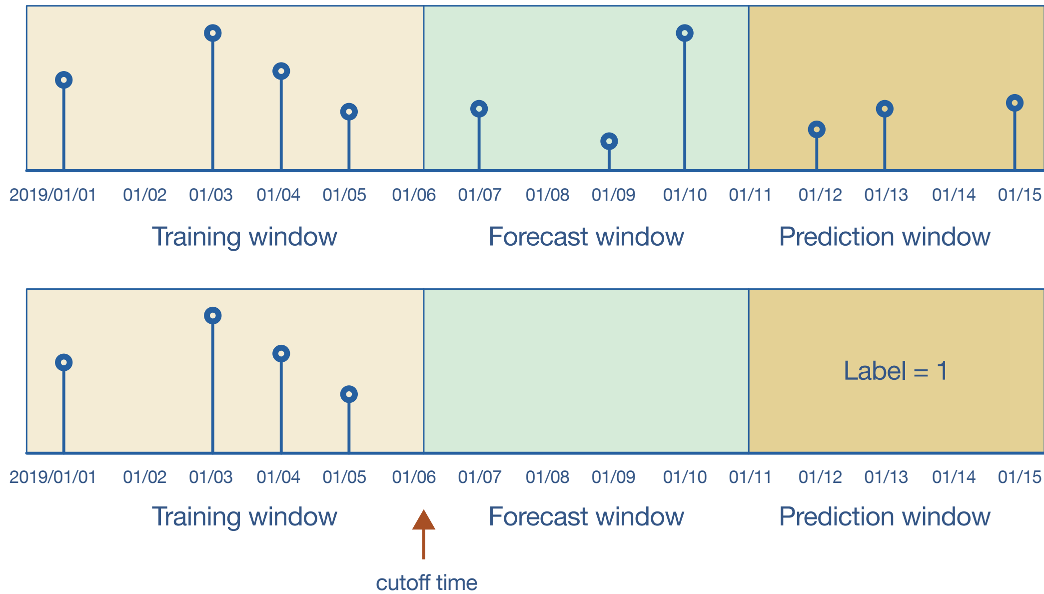



Figure 2-3: Visualization of prediction engineering concepts. The training, forecast, and prediction windows all have a length of five days, and the labeling function generates a target of 1.

### 2.4.2 Prediction Engineering for Wind Turbines

In this thesis, prediction engineering is applied to operations data from wind farms. This includes information about turbine stoppages, work orders, and notifications, as described below. All of these sources of historical data contain information that can be used to derive targets for a number of predictive problems. Although they are related, each source is made up of slightly different wind events that can be leveraged.

Turbine stoppages occur due to maintenance events, as well as high or low ambient wind speeds or other environmental restrictions. Stoppages can be either manual and automatic. The stoppages events data includes information about the cause, comments regarding the stoppage, the duration of the stoppage and the estimated generated power loss.

The work order data contains a summary of the maintenance work done on the turbines. Work order entries are usually generated by technicians in the field. Each

work order entry contains information on the work done, along with dates related to each step of the work order process. Similarly, notifications contain information about the parts that caused the stoppage, whether or not there was a breakdown, the priority of the incident, and other qualitative observations. Thus, each data source has event observations that, if extracted correctly with prediction engineering, provide important targets for the machine learning process.

## 2.5 Design Considerations

Because one goal of `GPE` is to enable wind domain experts to utilize prediction engineering, much of the design of this library focuses on usability. The design reflects the prioritization of the following goals, though others were considered as well.

1. **Flexibility.** `GPE` should work for multiple formats and types of operations data. We found that while our specific dataset only contained three files, other wind farms may have more. Thus, the framework should be able to ingest that. The users should have the ability, through a common interface, to define arguments that represent their changing machine learning needs.

2. **Robustness.** Though `GPE` should be flexible, it should also be safe to use. Users should know immediately when and why processing or target generation cannot be carried out. It should also be well-tested for a variety of scenarios so it can be safely applied in a industrial setting.

3. **Ease of use.** With the goal of enabling subject matter experts, the user interface must be easy to use. The library should provide the building blocks for experts to be able to define labeling functions beyond the ones that already exist in the library. They should also be able to easily choose desired parameters or call the methods without the need for parameter definition.

## 2.6 Library Overview

This section provides an overview of the `GPE` library and its various components. The library installation, the input data expectation, the components, and the user interaction are described. The library aims to enable subject matter experts to utilize the full prediction engineering process, from the merging of data sources to the generation of targets and cutoff times.

### 2.6.1 Project Structure

`GPE` is a Python package hosted on a GitHub repository that is pip-installable via a private PyPI package, as it is currently not open-source. The library has the components described below.

- `gpe` contains the code for processing the data and applying labeling functions to produce targets.
- `gpe.labeling` is a sub-package that contains both specific and generalized labeling functions for input into the target generation function.
- Configuration and make files.

### 2.6.2 Data Input Format

The library expects the aforementioned `stoppages`, `work orders`, and `notifications` data as inputs. Since one of our design priorities is input flexibility, the input allows for the following cases:

- Stoppages, work orders, and notifications, each within their own excel files.
- Work orders and notifications already merged, with stoppages in a separate file.
- Either stoppages and notifications only or stoppages and work orders only, with each as a separate file.

- Any of the combinations above, with an added file with additional data (e.g. parts replaced data).

There is flexibility not only in how the data is ingested, but in the format of the input as well. Most fields are flexible, as only those required for processing have their data types strictly checked. Additionally, if the required fields exist in the inputted data under a different name, the user is able to provide a mapping from the names in their data to the default names used by the module. Field and data type checking is again done on the labeling function level, but only for fields that are required for its application. The formats expected from the input excel files are as follows:

- **Stoppages**. Each raw entry must have a reference start and end timestamp, a turbine ID, and the duration of the stoppage.

- **Work Orders**. Each raw entry must contain the IDs of the work order and its associated notification. It must also contain the functional location of where the work order was carried out.

- **Notifications**. Similar to work orders, each raw entry must contain the IDs the notification and the associated work order, and the functional location of the notification. The entry must also contain the duration of the breakdown that caused the notification.

- **Additional Data**. Additional data must have the IDs of the turbine, work order, and notification. If any other data type requirements for the fields are inputted along with the data, those are checked as well.

## 2.7 Data Pre-processing

Given a combination of stoppages, work orders, notifications, and additional data, we first pre-process the data for target extraction. For the purposes of this section, we will refer to a data frame composed of merged work orders, notifications, and additional data as `wna`. Pre-processing involves two main steps: data parsing and initial processing, followed by data linking and merging.

### 2.7.1 Parsing and Initial Processing

Given the source file paths, the data is first extracted from the excel files. During the extraction process, date and time fields referring to same time measurement are merged to form a separate timestamp field. The required fields in the data are then checked. Finally, the duration of events in the data are unit standardized, and the turbine id is extracted from the functional location for the work orders and notifications.

### 2.7.2 Linking Data Tables

After the work orders, notifications, and additional data are parsed and processed, they are then merged by the IDs of the turbine, work order, and notification to produce the `wna` data frame. We then try to find a linkage between the stoppages and `wna` data. Linkages can only be found between data from the same turbine. If a valid linkage is found, it is recorded as a `stoppage id` for each `wna` entry. For simplicity, we decided to only allow a 1 to n correspondence for the linkage: a single stoppage to multiple work orders correspondence. The stoppages and `wna` are considered to be the 'parent' and 'child' datasets respectively.

In order to determine valid linkages, we first find condition-based linkages. Conditions are inputted by the user via a dictionary, and each condition includes the stoppage column, the `wna` column, and the keyword(s) that represent the condition. For example, if a condition included `Comments` as the stoppage column, `Malfunction information` as the `wna` column, and `yaw` as the keyword, then stoppage entries with `yaw` in the `Comments` column would be considered to have a condition-based linkage to `wna` entries with `yaw` in the `Malfunction information`. Condition-based linkages are determined as follows:

- If there is only a single keyword and a single condition, then the keyword must be present in the both of the identified columns.

- If multiple keywords are specified, any of the keywords can help create the linkage.

- If multiple conditions are provided, the output is the union of the condition-based linkages found.

- If no conditions are provided, then linkage determination relies on time overlap, which is described further below.

However, a linkage is only considered valid if both a condition-based and time-based linkage are found for a pair of entries. Users input the threshold they consider to be the minimum overlap threshold to qualify the stoppage and `wna` entry as linked. If the users do not give a threshold, then any time window overlap would be considered a time linkage. The time overlap is calculated based on the method defined in section 2.3.

### 2.7.3 Processing Output

If a merged output is requested, the pre-processing method returns all of the extracted data as a single data frame. The stoppages and `wna` are merged based on the linkages. If entries of the `wna` are not linked to stoppages, they will not appear in the output. Otherwise, if a merged output is not requested, the stoppages and the merged work orders, notifications, and additional data (or a subset depending on the input) are returned separately and in their entirety.

## 2.8 Target Generation

The pre-processed data contains turbine events that can be used to generate targets. The `GPE` library provides the functionality to generate target times from a single processed data frame. Target generation involves several components: a labeling function sub-package, the defining of prediction engineering parameters, integration with an external library called `Compose`, and target post-processing.

### 2.8.1 Labeling Functions

Through the library's labeling function sub-package, we provide users with functions to generate targets given a data slice. The defined labeling functions are also one of two types: specific labeling functions, or generalized functions used to generate other labeling functions. Specific labeling functions are labeling functions used to generate targets for one specific prediction problem — for example, the presence of a converter failure or the total power lost.

Generalized labeling functions, on the other hand, can generate new functions representing prediction problems. They use their inputs to produce specific functions applicable to a data slice. Current generalized labeling functions include those that search for keywords in specific columns, merge other labeling functions based on a connector, or calculate duration given a timestamp column. The keyword search function (`keyword_in_text`), for example, takes in a keyword and columns, and outputs a function that targets a data slice based on whether the specific keyword is in the designated columns.

```
def gear_presence(df):
  return keyword_in_text(
      'gear',
      columns=[
          'Comments',
          'Description'
      ]
  )(df)
```

```
def pump_presence(df):
  return keyword_in_text(
      'pump',
      columns=['Description']
  )(df)
```

Figure 2-4: Example usage of generalized labeling functions to create specific ones. We use `keyword_in_text` to create functions searching for the keywords *gear* and *pump* within designated columns.

Thus, generalized functions can be used as the building blocks for other functions. As more problems are defined and more labeling functions are needed, they can assist subject matter experts in adding specific functions to the labeling function sub-package.

### 2.8.2 Target Generation Function

The target generation function called by a user takes in data and a labeling function as its primary arguments. The data is checked for the required arguments and data types used by the labeling function. Each function, whether specific or created by a general function, has annotations about its arguments as an attribute. The labeling function is then applied on each slice of the input data. These data slices are generated using the prediction engineering parameters. The target of the data slice is either the output of the labeling function or a binary label created from setting a threshold on the output.

### 2.8.3 Prediction Engineering Parameters

When generating targets, we apply various prediction engineering parameters in order to better represent a problem or achieve an optimal result. Users provide these parameters through direct argument input or by loading them via a JSON file. The prediction engineering parameters are defined in Table 2.3, with the effects on the outputted data further explored in section 4.3.3.

| Parameter | Description |
|---|---|
| `target_entity` | entity on which to generate targets |
| `window_size` | size of the prediction window |
| `num_examples_per_instance` | number of examples per unique target entity value |
| `min_data` | minimum data or time before starting label generation |
| `gap` | time or number of entries between examples |
| `drop empty` | whether to drop the empty data slices |
| `threshold` | minimum numerical threshold for binary label generation |
| `lead` | forecast window size |

Table 2.3: Description of prediction engineering parameters.

### 2.8.4 Integration with Compose

In order to generate target times while appropriately applying prediction engineering parameters, methods from the `Compose`[3] library are used. The library provides tools for the automation of the prediction engineering process, based on the methods described in [14]. We initially attempted our own prediction engineering implementation; however, as `Compose` is open source and actively maintained, we decided instead to use it as the framework for our target generation interface. Specifically, `Compose` provides methods to generate data slices and associated cutoff times using the aforementioned prediction engineering parameters (Table 2.3).

## 2.9 GreenGuard

In this section we introduce the `GreenGuard`[4] library, which takes the output of `GPE` as an input. `GreenGuard` is an open-source Python package developed by DAI Lab members to create a pipeline framework for energy data. The library is built using `BTB`, `MLBlocks`, and `MLPrimitives`. With the `GreenGuard` framework, the user can either use the tested pipelines already created within the library or easily build new pipelines for machine learning. These pipeline files follow the `MLBlocks` pipeline format. They simply require a user to define primitives (eg. encoders, classifiers, etc.), initial parameters, input names, and output names. The framework provides the infrastructure to utilize the primitives, tune the hyperparameters, and execute the defined pipeline. `GreenGuard` takes two file types as inputs: `target times` and `readings`. The fields of the two types are shown in Table 2.4.

### 2.9.1 GreenGuard Process

For the native pipelines in `GreenGuard`, the two types of files are sourced from the outputs of `GPE` and the processed output of signal readings. `target times` is the output produced using `GPE`'s target generation functionality.

[3] https://github.com/FeatureLabs/compose
[4] https://d3-ai.github.io/GreenGuard/

| | Filename | |
|---|---|---|
| Fields | `readings` | `target times` |
| | `turbine id` | `turbine id` |
| | `signal id` | `cutoff time` |
| | `timestamp` | `target` |
| | `value` | |

Table 2.4: GreenGuard data input file format.

The processed signal readings make up the **`readings`**. These can either be inputted from a data frame or loaded from a collection of files. Each file in this collection contains the readings for a single turbine over a single month. **`readings`** become associated with **`target times`** if the timestamp of a reading falls within a target's training window — the window of data before its cutoff time. The size of the training window is specified by the **`GreenGuard`** user during loading.

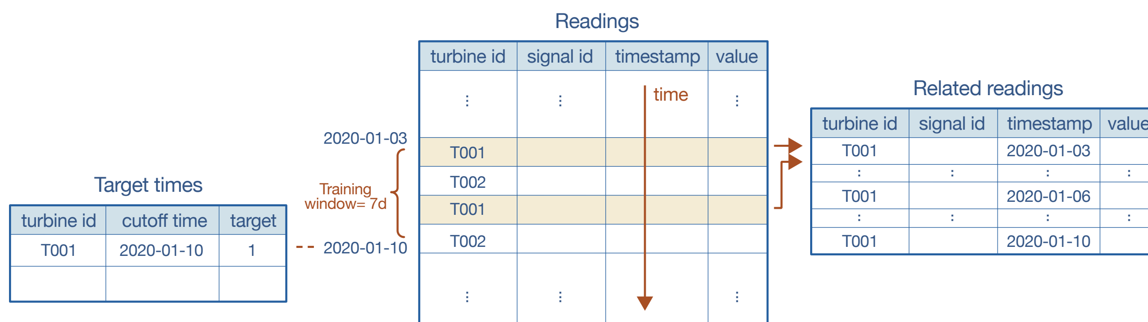



Figure 2-5: Relating the **`readings`** and **`target times`**. This diagram visualizes the process of extracting the training examples for each **`target`** based on the **`cutoff time`** and **`turbine id`**. The resulting data slice can then be passed to the machine learning model.

# Chapter 3

# Machine Learning Framework for Condition Monitoring Data

In this chapter we describe `CMS-ML`, a library for applying feature engineering and machine learning to Condition Monitoring System (CMS) data. The primary purpose of this library is to use vibration and other sensor data gathered by the CMS from sensors on different components to predict those components' failures. To do so, `CMS-ML` provides specific data processing techniques and an end-to-end modular framework for feature engineering and machine learning. This chapter goes over the motivations for `CMS-ML`, an overview of the library, and the details of its design and implementation.

## 3.1 CMS Data

Condition Monitoring Systems, as introduced in section 1.1, are vibration-based systems that gather data from sensors placed on important turbine parts. The CMS data in this thesis is specifically the output extracted from the Gram & Juhl turbine condition monitoring (TCM) system, which is installed in more than 80% of all offshore turbines [13].

The CMS data contains both signal readings and contextual information, ranging from the ambient wind speed to the units of the readings. The signal readings are often

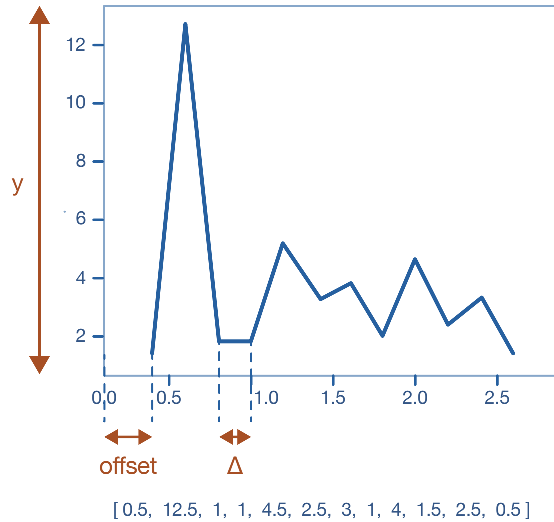



Figure 3-1: Example FFT spectrum with the `y-values`, `offset`, and `delta (Δ)`. The x-axis contains the frequency values, while the y-axis contains the amplitude values.

represented in the frequency domain as FFT values, although other representations may exist. In the CMS dataset, the FFT spectrum values are represented by three components:

- `y-values`. The y-axis values of the FFT spectrum.
- `offset`. The distance from the origin to the first value on the x-axis.
- `delta`. The constant distance between the x-values.

The x-values can be easily generated from or decomposed into the `offset` and `delta`. The three components are shown on spectrum in Figure 3-1. The FFT values also have associated context data, such as the `signal id`, `sensor name`, `turbine id`, and `timestamp`.

CMS data values are often represented as FFT values because vibrations due to turbine issues can be associated with particular frequencies, making the cause of the issue easier to discover [19]. In addition, anomalies can often be seen more easily in the frequency domain.

Much of the motivation behind the dedication of a specific machine library to CMS data comes from the fact that analysis of the spectral values can clearly provide

important insights toward predicting component issues. The CMS data also presents some unique challenges due to its nested data format and the pre-processing needs of the vibration signals, which are further detailed in this chapter.

## 3.2 Design Considerations

`CMS-ML` allows users to extract data and build pipelines for prediction problems using CMS data. Thus, the library's design is largely focused on bolstering the user experience. We sought a flexible, adjustable system where domain experts can easily create more pipelines beyond the scope of the ones already defined within the library. We focused on the following goals for the design of the `CMS-ML` library, although our library reflects other goals as well.

1. **Modularity and Abstraction.** Modularity is important in order for easily providing as many machine learning and feature engineering solutions as possible. Different parts of the pipeline can be joined, replaced, and adjusted as necessary. For simplicity, this is best done by creating a common abstract interface for all implemented methods.

2. **Efficiency.** Due to the large size of the CMS data (detailed in section 4.1.2), the `CMS-ML` methods must be both time and space efficient. Any unused data stored in memory should be effectively disposed of to ensure there is enough memory to process the CMS data.

3. **Ease of Use.** To best enable subject matter experts, `CMS-ML` should provide the user with an easy-to-use interface. Experts should be able to easily choose desired parameters for custom solutions and extend the current library.

## 3.3 Library Overview

This section provides an overview of the `CMS-ML` library and its various components. Specifically, we present how the library is installed, the expected data format, and

the contributions of each of its methods and components.

### 3.3.1 Project Structure

`CMS-ML`, like `GPE`, is a pip-installable package hosted on a private GitHub repository. It contains the following components:

- `cms_ml` contains the code for parsing the raw JSON inputs and applying functions to the processed data. It also contains **aggregation functions**, which are functions that group rows of data together to output a single summary value, and **transform functions** for the CMS data. These are considered the **primitives** of the CMS pipelines.
- `cms_ml.primitives` contains the JSON representations of the primitives.
- `cms_ml.pipelines` contains the JSON annotations of feature engineering and machine learning pipelines.
- Configuration and make files.

### 3.3.2 Data Input

The library expects JSON files containing the CMS data. Each JSON file contains context and vibration readings for a signal in a certain power bin range for a turbine. The processing function takes, as input, a path pointing to a folder of sub-folders, each containing the CMS JSONs for a single turbine. The JSONs themselves have a nested structure where different collections of information are grouped together.

Fields regarded as the context of the entry include many that originate from the SCADA system, which the CMS system is able to integrate with [6]. These fields include wind speed, yaw angle, gear oil temperature, and active power for determining the power bin [25]. The entry's dataset is given by the y-axis values of the vibration data, which has an associated x-axis offset and x-axis value delta.

### 3.3.3 Data Pre-processing

The data is processed per file per turbine, and contains only the specified turbines being processed. Since the files have an extremely nested structure, a data frame, containing the nested fields as columns, is first parsed from each JSON file. This frame is then merged with the rest of those parsed from the same turbine. This process is done separately for the context and the signal readings dataset due to the large size of the data. Since the context will contain the same information across all aggregations applied on the data, we reduce the memory usage and processing time by only generating the context once for each unique set of values. All fields except for `y-values` are extracted and returned as the context. For the dataset, only `signal id`, `timestamp`, and `y-values` are extracted. The `signal id` is composed of the signal name, sensor name, and applied function.

After either the context or the dataset is extracted, the extracted values are then filtered by the desired time range and the type of signal. The filtering at this step also helps reduce the size of the output. We also collect garbage during the processing steps in order to reduce the amount of memory used.

### 3.3.4 Primitives

#### Base Primitives

To help create CMS pipelines, we provide a library of functions for use as base primitives. These include single-value aggregation and signal transform primitives. New primitives can be easily contributed as well. The current aggregation and transform primitives were developed using feedback from discussions with subject matter experts and previous work in vibration analysis [3].

To use these methods in the development of the pipelines, each primitive has a corresponding JSON annotation. There are also JSON annotations for primitives from external modules that were deemed useful, from libraries including `PyWavelets` and `SciPy`. The JSON primitive annotations follow the standard format described in [22]: They contain metadata about the input and output, along with any hyper-

parameters. Current primitives with associated JSON annotations for single-value aggregations and signal transforms are described in Table 3.1 and 3.2 respectively.

| **Primitive** | **Description** |
|---|---|
| *Mean* | Average dataset value: $\frac{1}{N}\Sigma_{i=1}^{N} x_i$ |
| *RMS* | Root mean square: $\sqrt{\frac{1}{N}\Sigma_{i=1}^{N} x_i^2}$ |
| *Variance* | Measure of dispersion around the mean: $\frac{1}{N-DoF}\Sigma_{i=1}^{N}(x_i - \overline{x})^2$ |
| *STD* | Standard deviation, square root of variance: $\sqrt{\frac{1}{N-DoF}\Sigma_{i=1}^{N}(x_i - \overline{x})^2}$ |
| *Kurtosis* | Measure of the 'tailedness' of the data: $\frac{1}{N \cdot \sigma^4}\Sigma_{i=1}^{N}(x_i - \overline{x}^4)$ |
| *Skew* | Measure of the lack of symmetry: $\frac{1}{N \cdot \sigma^3}\Sigma_{i=1}^{N}(x_i - \overline{x}^3)$ |
| *Crest Factor* | Measure of peak extremity: $\frac{1}{RMS} \cdot x_{max}$ |
| *Identity* | Original dataset values: $x$ |

Table 3.1: Descriptions of aggregation primitives.

| **Primitive** | **Description** |
|---|---|
| *IFFT* | Inverse discrete Fourier Transform from FFT amplitudes |
| *DWT* | Discrete Wavelet Transform from a time series |
| *STFT* | Short Time Fourier Transform from a time series |
| *Welch* | Welch's power spectral density estimation from a time series |
| *Frequency Band* | Filters data for a particular frequency band |
| *Frequency Bin* | Divides the data for into frequency bins |

Table 3.2: Descriptions of transform primitives.

### Metaprimitives

In order to apply the primitives to the data, we utilize **metaprimitives**, which are in essence connectors between the data processing output and the primitives. Metaprimitives allow the application of aggregation or transform primitives by column or by row. Metaprimitives are necessary for a few reasons:

- The aggregation and transform primitives take a single array of values as an input. However, the outputs of the initial extraction of CMS data are large tables with thousands of rows containing dataset arrays. The primitive needs to be applied on the dataset array itself.

- If there are multiple frequency bands and bins, each band or bin has its own corresponding array of values, which is the input to the primitive.

- In addition, for the sake of reducing memory usage and output size, we only generate the dataset context once per unique value set. However, the context may be necessary for calculating the primitive result. For example, if the `x-value delta` is necessary for a signal transformation, we need to retrieve that from the context.

Metaprimitives can take in series, matrices, and data frames, and a user can select the desired column(s) for use if necessary. Metaprimitives ensure that all the necessary inputs are ingested by zipping any external columns to the dataset. Along with values from other data sources, metaprimitives also take in parameters for the primitives they are used with. For example, if a metaprimitive is used to calculate the standard deviation of each row, it can accept parameters such as degrees of freedom. A diagram showing the application is shown in Figure 3-2.

The metaprimitive is annotated by a JSON file in the same manner as a normal primitive. Within a pipeline, it effectively replaces the primitive, as the metaprimitive process handles creating and applying the block representing the primitive. Within the JSON annotation of a pipeline, the metaprimitive would be represented as demonstrated in Figure 3-3

### 3.3.5 Pipelines

The `CMS-ML` library contains custom primitives and pipelines for processing and running models on the CMS signal data. Though `MLBlocks` and `MLPrimitives` provide the framework for creating and running the pipelines, the `CMS-ML` library contributes

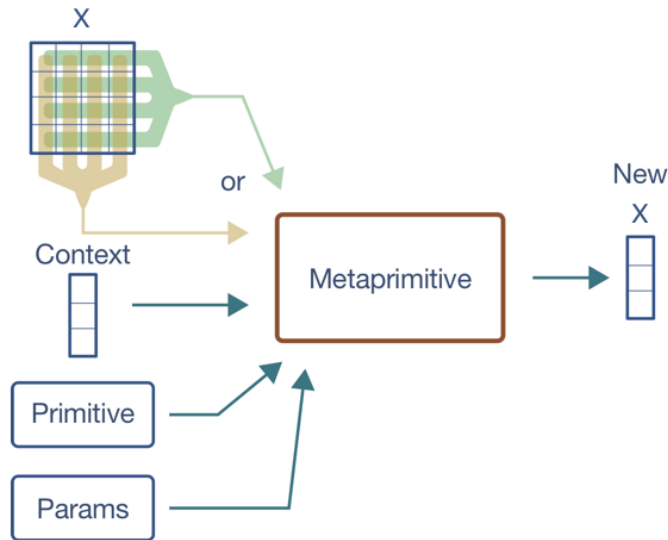



Figure 3-2: Metaprimitive usage. Metaprimitives can apply the primitive to either axis of the input matrix X, and can take in additional data and parameters.

```
{
    "primitives": [
        "cms_ml.meta.ApplyPrimitive",
    ],
    "init_params": {
        "cms_ml.meta.ApplyPrimitive#1": {
            "primitive_name":
            "cms_ml.statistics.crest_factor",
            "keywords": {"X": "values"},
            "axis": 1
        }
    }
}
```

Figure 3-3: Pipeline containing the metaprimitive which applies the primitive `cms_ml.statistics.crest_factor` to the data.

additional functionalities, as its pipelines include the primitives and metaprimitives for applying aggregations and transformations to the CMS data. Integrating the `CMS-ML` framework with existing pipeline technologies also gives the ability to tune the hyperparameters of the primitives, which can be used to get the best results for both the full machine learning process and the feature engineering output.

The `CMS-ML` pipelines can either be feature engineering pipelines, or machine learn-

ing pipelines that contain the feature engineering pipelines as sub-pipelines. The feature engineering pipelines generally follow the same workflow, which is shown in Figure 3-4. In a machine learning pipeline, the feature engineering output is then processed and encoded to develop a model using applicable `MLPrimitives`.

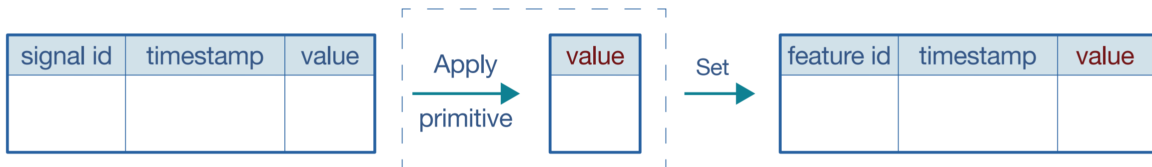



Figure 3-4: Feature engineering pipeline workflow. Within the dotted boundaries is the metaprimitive, which creates a chain with other metaprimitives if necessary.

The processed CMS data contributes signal readings for the pipelines to use; however, to model a prediction problem, the target times must also be provided. This can be done with prediction engineering, but for this thesis the target times and values are given directly by the subject matter experts, along with comments and explanations regarding the events. Results from exploring various pipelines for initial prediction problems are described in section 4.4.

## 3.4 User Interaction

Users are able to interact with the `CMS-ML` library in multiple ways. For feature engineering, they can directly call the Python function for aggregation application with the desired aggregation functions, which they can import or define to extract a specific output. An example of the direct function call is shown in Figure 3-5.

```
extract_cms_features(aggregations={"rms": cms_ml.statistics.rms,
                                   "mean": numpy.mean},
                     jsons_path='./data',
                     output_path='./output')
```

Figure 3-5: An example of directly calling the function that extracts features from the CMS data. The aggregations applied are `mean` and `rms`. The CMS data is read in from the `./data` folder. Since an `output path` is specified, the feature engineering result is written to a file in the `./output` folder and nothing is returned.

`CMS-ML` also has a command line interface to enable direct use of this function,

shown in Figure 3-6. Users can also build pipelines solely for feature engineering. These pipelines are made up of the aggregation and transform function primitives and metaprimitives further described in section 3.3.4.

```
(CMS-ML-venv) wang19k@dai-desk1:/mnt/users/wang19k$ cms_ml features -i data -o output -a mean -a rms -v
2020-01-26 22:26:39,584 - 27555 - INFO - __main__ - Extracting CMS Features:
2020-01-26 22:26:39,584 - 27555 - INFO - __main__ -     Input: data
2020-01-26 22:26:39,584 - 27555 - INFO - __main__ -     Output: output
2020-01-26 22:26:39,585 - 27555 - INFO - __main__ -     Aggregations: ['mean', 'rms']
2020-01-26 22:26:39,588 - 27555 - INFO - parsing - Parsing JSON file data/T001/Site 1 T001 2019-02-01 to 2020-01-31
Main bearing FFT.json
```

Figure 3-6: Command line interface with `mean` and `rms` as the aggregations. This command is equivalent to the Python function shown in Figure 3-5.

Users can also use `CMS-ML` to create prediction models based on the CMS data. This is done via tunable, end-to-end pipelines, which contain aggregation, transform, and machine learning primitives. Currently, the pipelines are created and run using the standard `MLBlocks` interface, shown in Figure 3-7. The pipeline expects, in the *fit* stage, the `turbine ids` and `cutoff times` of the training data, and the associated targets. It also expects readings in the form of a processed CMS data frame with an array of raw signal values per row. The *predict* stage expects the `turbine ids` and `cutoff times` of the testing data and readings in order to generate predictions. The `CMS-ML` library has tested pipelines ready for use, and new pipelines can be easily created.

```python
from mlblocks import MLPipeline

pipeline = MLPipeline('cms-ml-pipeline')
pipeline.fit(X_train, y_train, readings=readings)
predictions = pipeline.predict(X_test, readings=readings)
```

Figure 3-7: An example of using the `MLBlocks` interface to make predictions with a `CMS-ML` pipeline. `cms-ml-pipeline` is the name of the pipeline. `X_train` is a data frame containing the `turbine ids` and `cutoff times` of the training data and `y_train` is a vector containing the associated targets. `X_test` contains the `turbine ids` and `cutoff times` of the testing data. `readings` is the processed CMS data.

# Chapter 4

# Analysis and Evaluation

## 4.1 Data Overview and Analysis

### 4.1.1 Data Overview

The primary dataset contains data from four sources, describing 140 turbines from the same wind farm. All data from these sources have at least one timestamp and are associated with a turbine. The first data source is the operations data used as the input to the `GPE` library, as introduced in sections 2.4.2 and 2.6.2. Another data source is the vibrations signal data from the CMS systems, introduced in section 3.3.2 as the input to the `CMS-ML` software. The other two sources, OEM SCADA data and SCADA high temporal resolution data, are also primarily comprised of data readings. The signal readings within the SCADA data are currently being used by `GreenGuard` as inputs. Both SCADA datasets were extracted and processed from database access files into the following format:

| turbine_id | signal_id | timestamp | value |
|---|---|---|---|

### 4.1.2 Exploratory Analysis

After initial processing of the provided data, an exploratory analysis was conducted both to get a better understanding of the data and to verify with subject matter experts that, at first glance, it was correctly extracted. As introduced above, the data comes from four sources: CMS signal data, OEM SCADA 10-minute data and SCADA high temporal resolution data, and operations data records. Each row in the CMS data contains vibration and other turbine-related data, some of which is extracted from the SCADA data. The vibration data comes from several different sensors, including those on the generator and the main bearing. The CMS dataset contains FFT spectrums, cepstrums, and envelopes and is sampled irregularly.

The OEM SCADA data contains over 600 different types of signal, including component temperatures, wind speed, and power generation. These signals include those representing the mean, range, standard deviation, and end value of signals over a 10-minute interval. The SCADA high-resolution data on the other hand only has raw values and is irregularly sampled at a much higher frequency. The exploratory statistics found for each of the source types are shown in Table 4.1.

| data source | data points per turbine | date range |
|---|---|---|
| CMS | $300,000$ | 2015 - 2018 |
| OEM SCADA | $300,000,000,000$ | 2009 - 2018 |
| High Resolution SCADA | $50,000,000,000$ | 2015 - 2018 |
| Operations Data | $200,000$ | 2011 - 2018 |

Table 4.1: Exploratory statistics of the data sources.

## 4.2 Initial Prediction Problems

To evaluate the effectiveness of our framework, we define a few classes of initial prediction problems, which we can then apply prediction engineering to and build

machine learning models for. These problems were selected based on the preferences of subject matter experts.

### 4.2.1 Yaw Failures

Yaw systems are essential for the optimal functioning of wind turbines. Yaw systems typically consist of many moving parts, including yaw bearings, yaw drives (with yaw motors), yaw brakes, and other gears [16]. They ensure that, to maximize power input, the turbine is aligned with the incoming wind direction [16]. Thus, a yaw system failure can greatly affect power generation. Faulty yaw systems can also contribute to significant load fluctuations, potentially threatening the turbine's safety [26]. Based on the results of reliability studies, yaw systems are shown to be the second most common mechanical contributor to both the overall turbine failure rate and turbine downtime [27] (as cited in [16]). Once broken, they can also be difficult to repair [26]. As predictive and preventive maintenance for yaw systems is clearly an important problem, it was chosen as a prediction problem used to evaluate this work.

### 4.2.2 Gearbox Problems

The gearbox — which, along with the generator, makes up the drive train — is an essential part of most wind turbines' ability to produce electricity. The gearbox transfers shaft power from a lower-speed rotor to a higher-speed generator [20]. This is especially necessary in large turbines which rotate much more slowly. While the failure rate of gearboxes is generally not high, they have an important role in most turbine operations, so failures can cause significant downtime and have high maintenance costs [20].

Given the importance of gearboxes, previous works have been conducted using CMS vibration data for gearboxes, mainly for the diagnosis and early detection of potential issues. Some techniques used by previous work include the use of wavelet transform techniques, Empirical Mode Decomposition (EMD), and higher-order statistical techniques [21]. We try some of the signal processing techniques recommended

by previous works. Since we do not currently have confirmed gearbox failures for our data, we focus on the problem of predicting any general gearbox problem.

## 4.3 GPE Parameters

### 4.3.1 Processing Parameters

Choosing parameters for the pre-processing step of `GPE` can have an effect on the data used for generating targets, as shown in Table 4.2. The threshold and linkage conditions can impact how many work orders and notifications appear in the pre-processing output. Conditions that are too stringent can cause a loss of valuable information, while those that are too loose can link entries that are not actually connected. In addition, if different timestamps within the same data entry vary a lot, this can also affect the linkages.

### 4.3.2 Representation of Prediction Tasks

Prediction engineering parameters, especially the sizes of the training window, forecast window, and prediction window, can help represent different prediction tasks. Collaborating with wind subject matter experts, we defined three different tasks, translating their functional meanings into window size parameters. Note that applying the desired training window is not part of the prediction engineering process — it is instead done within the pipeline through time series processing primitives.

Using a labeling function that searched for yaw-related maintenance events within the notifications and work orders and data outputted by the `GPE` pre-processing function with the default arguments, we generated targets for each of these tasks. This was done using the maintenance start time as the time index, which is a reference time for the notifications data. We generated target times for a single turbine to show a clear example of the parameter effects. Table 4.4 gives an overview of the generated targets, while 4-1 provides a visualization of the target times.

Given the spread of the notification data for the selected turbine, there is not much

| threshold | conditions | reference timestamp | % of stoppages with linkages |
|---|---|---|---|
| 0.0 | — | Start of Malfunction<br>End of Malfunction | 12.56 |
| 0.5 | — | Start of Malfunction<br>End of Malfunction | 12.53 |
| 1.0 | — | Start of Malfunction<br>End of Malfunction | 12.01 |
| 0.0 | — | Start of Work Order<br>End of Work Order | 12.61 |
| 0.0 | — | Start of Notification Required<br>End of Notification Required | 12.48 |
| 0.0 | *gear* in Comments, Maintenance Information | Start of Malfunction<br>End of Malfunction | 0.64 |
| 0.0 | *gear* in Comments, Work Order & Notification Text Columns | Start of Malfunction<br>End of Malfunction | 0.69 |

Table 4.2: Effects of the `GPE` pre-processing parameters on the % of stoppages that correspond to a work order and notification.

visual difference between the first two prediction tasks. However, the timestamps are less shifted in the second prediction task, and there are slightly more targets due to the smaller data slices. In contrast, for the third prediction task, larger data slices mean there are fewer targets and the targets are more spread out and uniform.

### 4.3.3 Prediction Engineering Parameters

We also explored the effects of other prediction engineering parameters on the generated targets for yaw failure prediction, choosing to focus on the first predictive task (ie. normal or proactive maintenance). Tables 4-2, 4-3, and 4-4 display various statistics regarding how the prediction engineering output was affected by changing

| training window | forecast window | prediction window |
|---|---|---|
| 1. Normal or proactive maintenance planning | | |
| 1 week | 1 week | 1 week |
| 2. Unlimited technicians or short term interventions to influence availability | | |
| 3 day | 3 day | 3 day |
| 3. Batch maintenance or long term maintenance planning | | |
| 3 months | 0 day | 3 months |

Table 4.3: Problem statements for prediction tasks given by wind subject matter experts and their translation into corresponding parameters for training window, forecast window, prediction window.

| forecast window | prediction window | total targets | positive targets | first timestamp | last timestamp |
|---|---|---|---|---|---|
| 1 week | 1 week | 26 | 9 | `05-22-11 04:24` | `09-15-18 04:24` |
| 3 day | 3 day | 28 | 8 | `05-26-11 04:24` | `09-19-18 04:24` |
| 0 day | 3 months | 17 | 8 | `05-29-11 04:24` | `08-05-18 04:24` |

Table 4.4: Prediction task target time statistics.

the parameters.

Figure 4-2 shows the effects of varying the `min_data` parameter for a turbine, which specifies the amount of indices or time before we start searching for targets. Figure 4-3 presents the results from varying the `gap` between data slices, which can also be in terms of indices or time periods. The data clearly shows that the gap is important for properly spacing the targets and preventing or encouraging window overlap if necessary. Finally, in Figure 4-4, we use a numerical labeling function for the total kilowatt-hour power loss. The numerical threshold value that determines the binary output of the data directly affects the balance of positive and negative targets in the resulting targets.

Figure 4-1: Visualization of target times generated for the three prediction tasks. These targets represent the problem of predicting yaw-related maintenance events and are generated for a single turbine.

Figure 4-2: The effects of the `min_data` parameter on the generated target times for a single turbine. If `min_data` is an integer, it represents the number of entries before starting the search process. `min_data` can also be a period of time.

Figure 4-3: The effects of the `gap` parameter on the generated target times for a single turbine. If the `gap` is an integer, it represents the number of entries to use a gap. `gap` can also be a period of time.

## 4.4 CMS-ML Usage

In this section, we demonstrate the application of the `CMS-ML` pipelines with the initial prediction problems. We present statistics describing the feature engineering output

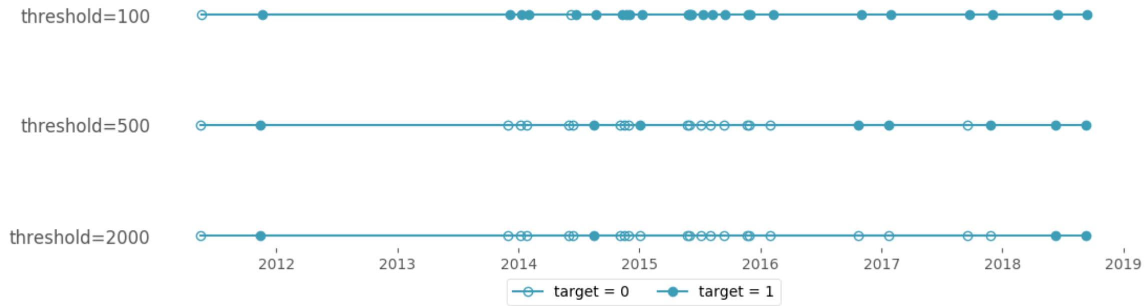



Figure 4-4: The effects of changing the `threshold` for converting the power loss output for a single turbine into a binary label.

and some preliminary results from running the selected machine learning pipelines. The models provide basic examples of the software's utilization and the impact of using different primitives, instead of as meaningful models for the prediction problems.

### 4.4.1 Data Preparation

To evaluate `CMS-ML`, we developed a couple of feature engineering and machine learning pipelines for use with the initial prediction problems. For the yaw prediction problem, we focused on predicting whether an expert-confirmed yaw gear failure would happen or not. For gearbox, since we did not have confirmed failures for the time range of the CMS data, we instead focused on predicting if any type of gearbox problem would occur. We represented these problems with targets that were manually extracted from the operations data.

Since the date range of the operations data (from 2011 to 2018) is larger than that of the CMS data (from mid-2015 to mid-2018), the targets first had to be filtered. In addition, because we were only given information about whether a problem occurred or not, we needed to pick some points in time that represented 'normal' operational circumstances. For simplicity, even though the non-problem to problem ratio is high, we sampled operational points until we had a balanced dataset. For the yaw gear failure and gearbox problem prediction problems, we used balanced, binary datasets that included 88 and 66 targets respectively.

### 4.4.2 Pipeline Details

The machine learning pipeline contained the following steps:

- After the `CMS-ML` primitives were applied to the data, the data was then resampled on a daily basis, imputed to fill in any missing values, and scaled.

- Then, using a training window of seven days, a time series matrix was extracted. This matrix consists of training examples, each represented by the signal values per each timestamp.

- Finally, the data was fitted to `keras.Sequential.LSTMTimeSeriesClassifier`, which was used to generate targets for new data.

Using `BTB`, we also ran each pipeline for five hyperparameter tuning iterations. An example JSON annotation of a full pipeline is shown in Appendix A, while a visualization of the pipeline's machine learning section is shown in Figure 4-5.

### 4.4.3 Results

To show the importance of primitive selection for the `CMS-ML` output, individual primitives and chains of primitives were tested. Due to the design of `CMS-ML`, the feature engineering process was simple. We were able to switch out primitives very easily by substituting the primitive name and initialization parameters within the metaprimitive block. We could also chain primitives by simply adding more consecutive metaprimitive blocks.

The statistics of the feature engineering output from the attempted primitives and primitive combinations are shown in Table 4.5. As shown, the choice of primitive(s) can impact the data value. In addition, as there are signals providing different types of measurements across many parts, the variance of the primitive output across the whole dataset can be relatively high.

The results from running the LSTM time series classifier with primitives for both the yaw gear failure and the gearbox problem are shown in Table 4.6. The values presented are the best results out of five attempts per pipeline. Though the results are

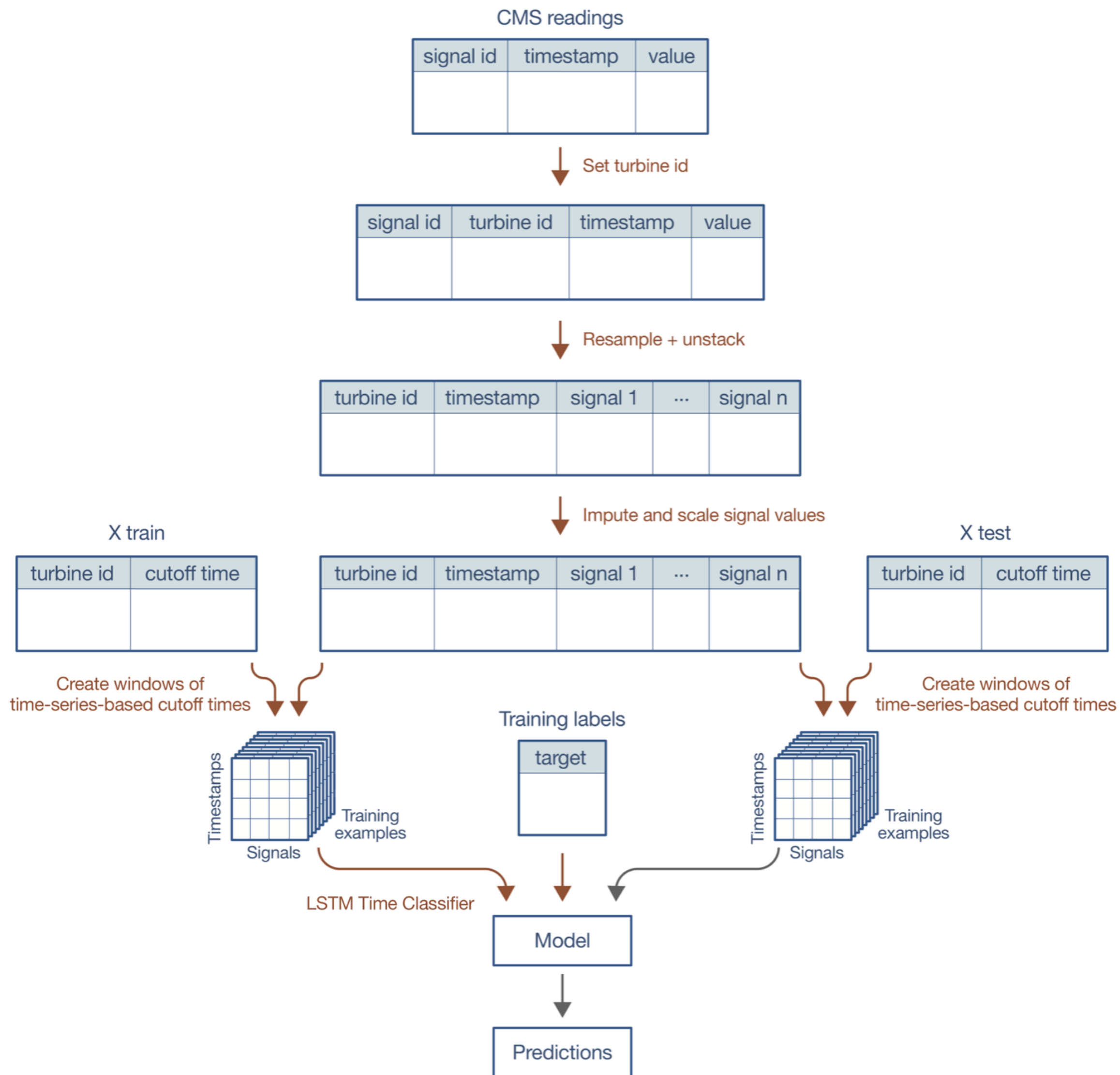



Figure 4-5: Machine learning pipeline used to generate predictions from CMS data.

not strong, they demonstrate the valid usage of `CMS-ML` pipelines to create end-to-end machine learning solutions using the CMS data.

## 4.5 Concluding Remarks

In this section, we demonstrated the usages of `GPE` and `CMS-ML`. For `GPE`, we have demonstrated how parameters can be used to portray prediction tasks. The selection of `GPE`'s parameters, both during pre-processing and target generation, have a significant impact on how past data is represented. Though not shown, this in turn affects

| primitive(s) | mean | std | min | max |
|---|---|---|---|---|
| mean | 11.2 | 122.6 | 0.0 | 1560.6 |
| rms | 11.4 | 122.6 | 0.0 | 1560.7 |
| kurtosis | 277.4 | 319.5 | -3.0 | 4291.5 |
| variance | 17.6 | 329.6 | 0.0 | 162390.3 |
| variance ∘ Daubechies 2 wavelet | 52.8 | 1007.0 | 0.0 | 325472.4 |
| variance ∘ Daubechies 2 wavelet ∘ IFFT | 5.7 | 66.9 | 0.0 | 1012.9 |

Table 4.5: Statistics on the feature engineering output for various parameters.

| primitive(s) | problem | accuracy | F1 score |
|---|---|---|---|
| mean⌢rms | yaw | 0.59 | 0.64 |
| | gearbox | 0.59 | 0.36 |
| kurtosis | yaw | 0.63 | 0.63 |
| | gearbox | 0.53 | 0.55 |
| variance | yaw | 0.59 | 0.61 |
| | gearbox | 0.53 | 0.64 |
| variance ∘ Daubechies 2 wavelet | yaw | 0.55 | 0.69 |
| | gearbox | 0.41 | 0.44 |
| variance ∘ Daubechies 2 wavelet ∘ IFFT | yaw | .64 | .67 |
| | gearbox | 0.59 | 0.46 |

Table 4.6: `CMS-ML` prediction results. These metrics are not representative of what can be achieved once we refine the targets and optimize the machine learning pipeline. This is purely meant to demonstrate how the software works.

the models produced by `GreenGuard`. This work clearly motivates a future need to explore parameter selection in the context of wind machine learning models.

As mentioned previously, although we did not produce meaningful results from running the full `CMS-ML` pipelines, we were still effective in demonstrating the ease of their usage and the effects of primitive decisions. In addition, end-to-end tuning could be a promising advantage of this pipeline-based infrastructure. With further

exploration, more data, and better targets, we anticipate that this work can serve as a base for exploring increasingly better machine learning pipelines based on the CMS data.

# Chapter 5

# Conclusion

## 5.1 Future Work

Though great progress has been made towards developing a complete machine learning solution for wind turbine predictive maintenance, there are clear steps that should be taken to improve this project. The two immediate steps are detailed in the next subsection.

### 5.1.1 Improvement of Pipelines

The pipelines shown in this thesis showcase the usage of the wind machine learning libraries and how manageable they are to create and modify. While conducting this work and creating demonstrative pipelines, we identified various avenues of exploration to improve both the performance and usage of the pipelines. We focused on the following:

- The output of the pipelines is fairly reliant on the feature engineering performed by the aggregation and transform primitives. In our initial exploration, we tested the effects of a small number of primitives and combinations of primitives. Within the library, we also provided the primitives for additional signal transformations and aggregations. Further work should be done towards creating and testing more primitives and combinations for CMS feature engineering.

Given the feedback from the subject matter experts, we believe that the frequency band and bin primitives could be of particular interest. In addition, the models themselves can be further optimized or exchanged for different time series models.

- The process, given the size of the CMS data, can be fairly slow and memory consuming. We have made steps towards solving this through garbage collection and code optimization. However, further adjustments can be made, including signal filtering for data size reduction and threading. Signal filtering could leverage domain knowledge by obtaining insights from the experts about the importance of certain signals. A similar problem exists with the OEM and High Temporal Resolution SCADA data.

### 5.1.2 GPE User Interface

When users currently interface with `GPE`, they can easily generate sets of targets with a simple function call. However, we recognize that they must input prediction engineering parameters based on their best guess as to what they should be. They also need to know the name and usage of the labeling function that corresponds with the problem they are aiming to solve. This means that users may need some prior knowledge about how prediction engineering works and how the library has been implemented. Thus, `GPE` needs a better interface where users can easily explore both the parameters and labeling functions. This solution could take the form of a lightweight web app built with a REST API.

We also want to conduct user testing of `GPE` to get a better understanding of expert interaction and to iteratively improve the interface. We want to get feedback on how usable and learnable the interface is, and how well it suits their needs for prediction engineering.

### 5.1.3 Direct Integration

The current implementation of the wind machine learning system has been designed with integration and flexibility in mind. However, these libraries were created to ingest offline outputs from industry systems. Since a goal is for these systems to have significant exposure and be used towards solving a breadth of prediction problems, the data inputs and outputs of these systems must be easily integrable with existing databases. With this integration, there may be other priorities that need to be considered, such as security and data access controls.

# Appendix A

# Pipelines

```
{
    "primitives": [
        "pandas.DataFrame.pop",
        "cms_ml.meta.ApplyPrimitive",
        "pandas.DataFrame.set",
        "pandas.DataFrame.resample",
        "pandas.DataFrame.unstack",
        "pandas.DataFrame.pop",
        "pandas.DataFrame.pop",
        "sklearn.impute.SimpleImputer",
        "sklearn.preprocessing.MinMaxScaler",
        "pandas.DataFrame",
        "pandas.DataFrame.set",
        "pandas.DataFrame.set",
        "mlprimitives.custom.timeseries_preprocessing.cutoff_window_sequences",
        "keras.Sequential.LSTMTimeSeriesClassifier"
    ],
    "init_params": {
        "pandas.DataFrame.pop#1": {"item": "value"},
        "cms_ml.meta.ApplyPrimitive#1": {
            "primitive_name": "scipy.stats.kurtosis",
            "keywords": {"X": "a"},
            "axis": 1
        },
        "pandas.DataFrame.set#1": {"key": "value"},
        "pandas.DataFrame.resample#1": {
            "rule": "1D",
            "on": "timestamp",
            "groupby": ["turbine_id", "signal_id"],
            "aggregation": "mean",
            "reset_index": false
        },
        "pandas.DataFrame.unstack#1": {"level": "signal_id", "reset_index": true},
        "pandas.DataFrame.pop#2": {"item": "turbine_id"},
        "pandas.DataFrame.pop#3": {"item": "timestamp"},
        "sklearn.preprocessing.MinMaxScaler#1": {"feature_range": [-1, 1]},
```

```
        "pandas.DataFrame#1": {},
        "pandas.DataFrame.set#2": {"key": "turbine_id"},
        "pandas.DataFrame.set#3": {"key": "timestamp"},
        "mlprimitives.custom.timeseries_preprocessing.cutoff_window_sequences#1": {
            "window_size": 7,
            "cutoff_time": "cutoff_time",
            "time_index": "timestamp"
        },
        "keras.Sequential.LSTMTimeSeriesClassifier": {
            "epochs": 35,
            "input_shape": [7, 108]
        }
    },
    "input_names": {
        "pandas.DataFrame.pop#1": {"X": "readings"},
        "cms_ml.meta.ApplyPrimitive#1": {"X": "values"},
        "pandas.DataFrame.set#1": {"X": "readings", "value": "values"},
        "pandas.DataFrame.resample#1": {"X": "readings"},
        "pandas.DataFrame.unstack#1": {"X": "readings"},
        "pandas.DataFrame.pop#2": {"X": "readings"},
        "pandas.DataFrame.pop#3": {"X": "readings"},
        "sklearn.impute.SimpleImputer#1": {"X": "readings"},
        "sklearn.preprocessing.MinMaxScaler#1": {"X": "readings"},
        "pandas.DataFrame#1": {"X": "readings"},
        "pandas.DataFrame.set#2": {"X": "readings", "value": "turbine_id"},
        "pandas.DataFrame.set#3": {"X": "readings", "value": "timestamp"},
        "mlprimitives.custom.timeseries_preprocessing.cutoff_window_sequences#1": {
            "timeseries": "readings"
        }
    },
    "output_names": {
        "pandas.DataFrame.pop#1": {"item": "values"},
        "cms_ml.meta.ApplyPrimitive#1": {"X": "values"},
        "pandas.DataFrame.resample#1": {"X": "readings"},
        "pandas.DataFrame.unstack#1": {"X": "readings"},
        "pandas.DataFrame.pop#2": {"item": "turbine_id"},
        "pandas.DataFrame.pop#3": {"item": "timestamp"},
        "sklearn.impute.SimpleImputer#1": {"X": "readings"},
        "sklearn.preprocessing.MinMaxScaler#1": {"X": "readings"},
        "pandas.DataFrame#1": {"X": "readings"}
    }
}
```

Example of a full CMS-ML pipeline.

# Bibliography


[1] Y. Amirat, M.E.H. Benbouzid, E. Al-Ahmar, B. Bensaker, and S. Turri. A brief status on condition monitoring and fault diagnosis in wind energy conversion systems. *Renewable and Sustainable Energy Reviews*, 13(9):2629 – 2636, 2009.

[2] Yoshua Bengio, Aaron C. Courville, and Pascal Vincent. Unsupervised feature learning and deep learning: A review and new perspectives. *CoRR*, abs/1206.5538, 2012.

[3] Wahyu Caesarendra and Tegoeh Tjahjowidodo. A review of feature extraction methods in vibration-based condition monitoring and its application for degradation trend estimation of low-speed slew bearing. *Machines*, 5(4), 2017.

[4] B. Chen, P. C. Matthews, and P. J. Tavner. Automated on-line fault prognosis for wind turbine pitch systems using supervisory control and data acquisition. *IET Renewable Power Generation*, 9(5):503–513, 2015.

[5] P. Chen and M. Kezunovic. Fuzzy logic approach to predictive risk analysis in distribution outage management. *IEEE Transactions on Smart Grid*, 7(6):2827–2836, Nov 2016.

[6] C.J. Crabtree, D. Zappala?, and P.J. Tavner. Survey of commercially available condition monitoring systems for wind turbines. Technical report, DU, May 2014.

[7] Katharina Fischer and Diego Coronado. Condition monitoring of wind turbines: State of the art, user experience and recommendations. *VGB PowerTech*, 2015:51–56, 07 2015.

[8] Laura Gustafson. Bayesian tuning and bandits: An extensible, open source library for automl. M. eng thesis, Massachusetts Institute of Technology, Cambridge, MA, June 2018.

[9] Justin Heinermann and Oliver Kramer. Machine learning ensembles for wind power prediction. *Renewable Energy*, 89:671 – 679, 2016.

[10] R. L. Hu, K. Leahy, I. C. Konstantakopoulos, D. M. Auslander, C. J. Spanos, and A. M. Agogino. Using domain knowledge features for wind turbine diagnostics. pages 300–307, Dec 2016.

[11] R.L. Hu, J. Granderson, D.M. Auslander, and A. Agogino. Design of machine learning models with domain experts for automated sensor selection for energy fault detection. *Applied Energy*, 235:117 – 128, 2019.

[12] A. Jaech, B. Zhang, M. Ostendorf, and D. S. Kirschen. Real-time prediction of the duration of distribution system outages. *IEEE Transactions on Power Systems*, 34(1):773–781, Jan 2019.

[13] Gram & Juhl. Gram & juhl's tcm aftermarket solutions for the scottish cms market. 09 2018.

[14] James Max Kanter, Owen Gillespie, and Kalyan Veeramachaneni. Label, segment, featurize: a cross domain framework for prediction engineering. 2016.

[15] Udayan Khurana, Horst Samulowitz, and Deepak S. Turaga. Feature engineering for predictive modeling using reinforcement learning. *CoRR*, abs/1709.07150, 2017.

[16] M-G Kim and P H Dalhoff. Yaw systems for wind turbines – overview of concepts, current challenges and design methods. 524:012086, jun 2014.

[17] Kevin Leahy, Colm Gallagher, Ken Bruton, Peter O'Donovan, and Dominic T.J. O'Sullivan. Automatically identifying and predicting unplanned wind turbine stoppages using SCADA and alarms system data: Case study and results. *Journal of Physics: Conference Series*, 926:012011, nov 2017.

[18] Kelvin Lu. Feature engineering and evaluation in lightweight systems. M. eng thesis, Massachusetts Institute of Technology, Cambridge, MA, June 2019.

[19] Jason Mais. Spectrum analysis: The key features of analyzing spectra. *SKF USA Inc.*, 05 2002.

[20] S. McFadden and B. Basu. 7 - wind turbine gearbox design with drivetrain dynamic analysis. In Chong Ng and Li Ran, editors, *Offshore Wind Farms*, pages 137 – 158. Woodhead Publishing, 2016.

[21] Mengyan Nie and Ling Wang. Review of condition monitoring and fault diagnosis technologies for wind turbine gearbox. *Procedia CIRP*, 11:287 – 290, 2013. 2nd International Through-life Engineering Services Conference.

[22] Micah J. Smith, Carles Sala, James Max Kanter, and Kalyan Veeramachaneni. The machine learning bazaar: Harnessing the ml ecosystem for effective system development. *arXiv e-prints*, page arXiv:1905.08942, 2019.

[23] Yangqiu Song and Dan Roth. Machine learning with world knowledge: The position and survey. *CoRR*, abs/1705.02908, 2017.

[24] Peng Sun, Jian Li, Junsheng Chen, and Xiao Lei. A short-term outage model of wind turbines with doubly fed induction generators based on supervisory control and data acquisition data. *Energies*, 9(11), 2016.

[25] Liang Tao, Qian Siqi, Yingjuan Zhang, and Huan Shi. Abnormal detection of wind turbine based on scada data mining. *Mathematical Problems in Engineering*, 2019:1–10, 08 2019.

[26] Shuting Wan, Lifeng Cheng, and Xiaoling Sheng. Effects of yaw error on wind turbine running characteristics based on the equivalent wind speed model. *Energies*, 8(7):6286–6301, Jun 2015.

[27] M Wilkinson and B Hendriks. Reliawind – report on wind turbine reliability profiles tech. rep. gh. 2011.

[28] Wenxian Yang, P.J. Tavner, Christopher Crabtree, Yanhui Feng, and Yingning Qiu. Wind turbine condition monitoring: Technical and commercial challenges. *Wind Energy*, 17, 05 2014.