Enhancing a Data-Centric Framework for Predictive Maintenance of Wind Turbines

by

Raymond Pan

S.B. Computer Science and Engineering, MIT (2025)

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

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2025

©2025 Raymond Pan. All rights reserved.

The author hereby grants to MIT a nonexclusive, worldwide, irrevocable, royalty-free license to exercise any and all rights under copyright, including to reproduce, preserve, distribute and publicly display copies of the thesis, or release the thesis under an open-access license.

Authored by: Raymond Pan
Department of Electrical Engineering and Computer Science
May 2025

Certified by: Kalyan Veeramachaneni
Principal Research Scientist
Thesis Supervisor

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

Enhancing a Data-Centric Framework for Predictive Maintenance of Wind Turbines

by

Raymond Pan

Submitted to the Department of Electrical Engineering and Computer Science
on May 2025, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Predictive maintenance of wind turbines is a machine learning task aimed at minimizing repair costs and improving efficiency in the wind turbine and renewable energy industry. Existing machine learning solutions often fail to meet real-world deployment requirements due to fragmented pipelines, lack of domain integration, and reliance on black-box models. Zephyr, a data-centric machine learning framework, addresses these challenges by enabling Subject Matter Experts (SMEs) to incorporate their domain knowledge into the prediction process, and to leverage automated tools for labeling, feature engineering, and prediction tasks without requiring extensive technical knowledge. However, the current version of Zephyr still has limitations, including usability gaps and a reliance on external tools for certain steps.

Case studies with real-world data from the renewable energy company Iberdrola demonstrate Zephyr's potential to integrate domain expertise into wind turbine predictive maintenance (thus streamlining the process) but also expose a sub-optimal user experience. This thesis explores gaps in the current state of the Zephyr framework and proposes refinements to enhance its usability. Key improvements include the consolidation of current tooling and relevant external libraries into a single API, state management with careful logging and exception handling, and improved support for model evaluation. These enhancements aim to support seamless end-to-end predictive modeling workflows, and to provide a more refined and flexible user experience for the Zephyr user base.

Acknowledgments

I want to start by thanking my advisor, Kalyan Veeramachaneni, for his guidance and support throughout the project. I also want to thank everyone I met and worked with in the Data-to-AI lab over the last 2 years, especially Sarah Alnegheimish and Sara Pidó. Additionally, I want to thank my friends and family for being my friends and family.

I would also like to acknowledge Frances R. Hartwell for building much of Zephyr's foundation. Their research has been very instrumental in shaping my work.

Finally, I want to say that I really enjoyed my time at MIT – it has been four great years of learning, in and out of the classroom.

Contents

List of Figures

List of Tables

Code Listings

Chapter 1

Introduction

1.1 Overview

Zephyr is a data-centric machine learning framework designed to streamline model building for predictive maintenance of wind turbines. It was specifically designed to address the shortcomings of previous software systems [3]. Although several machine learning methods aimed at predicting wind turbine component failures have been developed, most of these exist only in the form of published research studies and have not been deployed in the real world. Further, models developed by ML engineers alone can be impractical for real-world use, because ML engineers often lack the critical domain knowledge and context required to make deployable models.

In this context, Zephyr was designed to be a best-of-both-worlds solution: By automating many of the tasks involved in the ML pipeline (such as data handling, labeling, and feature generation), Zephyr allows Subject Matter Experts (SMEs) to inject domain knowledge at key stages while utilizing the power of ML tooling and systems.

In particular, Zephyr aims to address two main concerns that have arisen with similar projects:

1. Piecing together diverse tools and integrating data can result in a significant amount of customized code infrastructure that must be maintained, which can

be overwhelming to the user – an issue known as the "pipeline jungle."

2. Black box solutions may not provide affordances for incorporating domain knowledge, making it difficult for users to understand – and therefore trust – the tool. Other tools that are more flexible are often hard for non-machine learning experts to use. Both prevent extremes subject matter experts (SMEs) from generating models that are usable in real-world scenarios.

In its current state, Zephyr [3] includes the following main features:

- **EntitySet creation**: Functions to create EntitySets, a data structure used to represent a collection of entities and the relationships between them, for datasets that use `SCADA`, `PI`, and/or `Vibrations` data (standard data sources within the wind industry).

- **Labeling Functions**: A collection of functions that can search past operations data for occurrences of specific event types (as well as tools to create custom versions of these functions). Said events, and the data preceding them, can be used to train a model that predicts future events.

- **Feature Engineering:** A guide to using Featuretools [5], a framework used to perform automated feature engineering, on wind farm data.

- **Prediction Engineering:** A flexible framework designed to apply labeling functions to wind turbine operations data in a number of different ways, in order to create labels for custom machine learning problems [4].

To showcase Zephyr's ability to integrate specific domain knowledge, we worked with Iberdrola, a renewable energy company that works with on- and offshore wind turbines. Iberdrola relies on automated tools and machine learning methods to predict when turbine failures might occur, making Zephyr a useful candidate.

While case studies performed using Zephyr with Iberdrola's data proved the effectiveness of Zephyr, we noticed that it felt slightly fragmented, requiring the user to import different Zephyr functionalities and manage intermediate outputs as they go

from step to step. In addition, Zephyr was unable to solve machine learning problems end-to-end without the incorporation of additional tools.

This thesis investigates the current limitations of Zephyr, and improves it as a tool for wind turbine predictive maintenance by refining its library. The following steps were performed:

1. Consolidated Zephyr's current tooling and relevant external libraries to reduce fragmentation.

2. Implemented state management for user's intermediate outputs while preventing "pipeline jungles" [6] and user confusion for end-to-end predictive maintenance workflows.

3. Generalized support for model evaluation, allowing users full control and flexibility over the input and output space for evaluation metrics.

1.2 Thesis Organization

This thesis is organized as follows: Chapter 2 introduces the current state of Zephyr and important concepts via a case study. Chapter 3 addresses the flaws introduced in Chapter 2 and details our implementation of our improvement to Zephyr. Chapter 4 revisits the case study from Chapter 2 using our new API to demonstrate and evaluate our new implementation of Zephyr. Finally, Chapter 5 summarizes the conclusions and details future work on the project.

Chapter 2

Background

To introduce Zephyr and the general predictive modeling workflow, we present a tutorial-style case study focused on predicting brake pad failures. This case study explores the current state of Zephyr, a ML library providing prediction engineering methods for wind turbine maintenance[1]. Table 2.1 summarizes key concepts and definitions relevant to Zephyr that will be introduced in this chapter. Table 2.2 lists the python libraries that Zephyr relies on.

By automating many of the tasks involved in the ML pipeline, such as data handling, labeling, and feature generation, Zephyr allows users to incorporate their domain knowledge at key stages, which results in more accurate models. For this section, assume that all the code listings in this section are run sequentially as they appear, as in a `Jupyter Notebook`, and that, therefore, every subsequent code listing may depend on any previous ones in this section[2].

2.1 Brake Pad Tutorial

In this tutorial case study, we will be trying to predict premature brake pad wear in turbines by utilizing data from 70 turbines and the Zephyr framework. The tu-

[1]Although Zephyr can be used in many other domains, it was designed for the wind industry.

[2]Note that this section focuses on the current version, of the Zephyr framework and uses the tooling currently provided as of Fall 2024, which may or may not be the same in any future Zephyr frameworks

Item	Definition
EntitySet	A collection of dataframes and the relationships between them
LabelTimes	The data frame that contains labels and cutoff times for the target dataframe. The process of generating *LabelTimes* from time-series data is governed by a set of parameters: `minimum data`, `cutoff time`, `prediction window`, and `gap`.
Feature Matrix	A tabular representation of feature values where each row represents an observation, and each column represents a feature.
Accuracy	The proportion of all classifications that the model made correctly
Recall	The proportion of actual positives that were correctly classified as positives
Precision	The proportion of all positive classifications that are truly positive.
F1 Score	The harmonic mean of precision and recall. It assesses a modelâĂŹs ability to maintain both high precision and high recall.
Confusion Matrix	Shows how many samples from each label were predicted correctly.
AUC-ROC Curve	The ROC curve is a plot of the True Positive Rate (TPR) against the False Positive Rate (FPR) at different classification thresholds. AUC is the area under the ROC curve, and measures how well the model can distinguish between two classes.

Table 2.1: Zephyr Key Concepts

torial will illustrate how Zephyr enables users to build a model that reduces turbine maintenance costs.

To provide context, wind turbine brake pads wear out over time, and premature wear leads to costly unscheduled maintenance. Efficient management of brake pad replacement requires periodic physical measurements, which are both costly and risky for maintenance personnel to obtain. However, if premature brake pad failure can be accurately predicted using signal and operations data, maintenance costs can be reduced significantly.

Library	Purpose
`Featuretools`	Framework to perform automated feature engineering
`composeml`	Machine learning tool for automated prediction engineering
`xgboost`	Optimized distributed gradient boosting library
`scikit-learn`	Simple and efficient tools for predictive data analysis
`MLBlocks`	Framework for seamlessly combining any possible set of Machine Learning tools developed in Python
`seaborn`	Data visualization library
`matplotlib`	Data visualization library

Table 2.2: Relevant Libraries.

2.1.1 General Predictive Modeling Workflow

Before jumping into Zephyr, let's lay out the steps needed to build an ML model that predicts premature brake pad failure.

A typical workflow for building predictive maintenance models involves the following steps:

1. **Loading & Pre-processing Data**: We load the data – which can consist of one or many tables – and then pre-process it.

2. **Labeling & Generating *LabelTimes***: We find previous instances of premature brake pad failure, and opposite examples where brake pad failure did not occur.

3. **Feature Engineering**: We use the examples from the data to extract features, so that we can train the model.

4. **Modeling**: We train a model using the extracted features and explore the model's hyperparameter settings.

5. **Evaluation**: After building the model, we evaluate whether it performs satisfactorily.

We then keep iterating over steps 3-5 until we achieve an effective model. Zephyr's goal is to make this workflow seamless, allowing users to rely on automation as well as their expertise to build this model.

We will now explore how Zephyr completes each of these steps.

2.1.2 Loading & Pre-processing Data

What is Data Pre-processing? Data pre-processing is a critical step in machine learning, as it allows us to clean and organize our data into a format our subsequent steps in building a model can understand and learn from. For wind turbines, there are two primary sources of data: signal data and operations data. The first step in the Zephyr framework is to combine both signal and operations data into a well-formatted, relational data structure. Much of this data is standard across wind turbines and farms, which we use to our advantage when ingesting it.

EntitySets

EntitySets are the structured, usable format that Zephyr uses. An *EntitySet* is a data structure made up of a collection of entities and the relationships between them. Figure 2-1 shows a high-level representation of a Zephyr *EntitySet* and its interconnected tables, which are useful for preparing raw, structured datasets for feature engineering. The current state of Zephyr provides functions that help create *EntitySets* from raw wind turbine data. In order for Zephyr to process the data correctly and represent it in an *EntitySet*, we must follow specific input data rules. We start by introducing the different types of data Zephyr expects.

Signal Data Signal data is time series data generated by wind turbine assets and their auxiliary systems. There are three primary sources of signal data that are standard in the industry:

- Original Equipment Manufacturer Supervisory Control and Data Acquisition (OEM-**SCADA**)

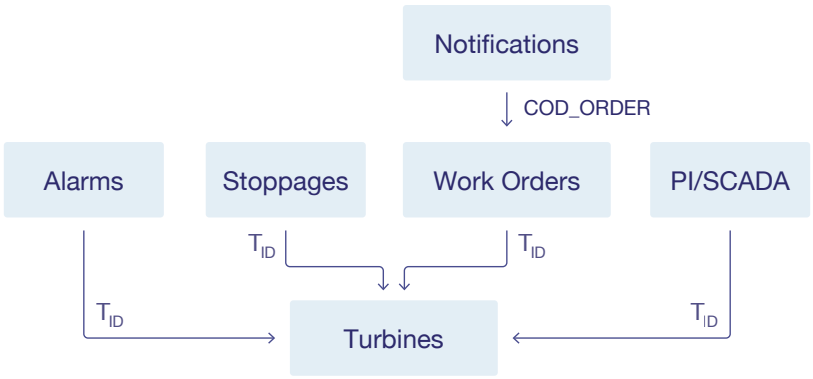


Figure 2-1: *EntitySet* Structure. Modified from [3]. The arrows indicate column values in the child *DataFrame* linked to the the index of the parent *DataFrame*, *Turbines*. The arrows from the child *DataFrames* to the parent *DataFrame* represent a many-to-one relationship. T_{ID} is the unique turbine identifier and COD_ORDER is the unique work order identifier.

- Historical Plant Information (**PI**)
- Vibration data collected on Planetary gearboxes in turbines (**Vibrations**).

At each timestamp, these systems provide various variables and signals, providing critical insight into turbine performance and operation.

Operations Data Operations data encompasses several categories of data collected during the routine operations and maintenance of a wind farm, including:

- **Turbines**: Records of each physical wind turbine and the attributes that identify and characterize each machine.
- **Alarms**: Information about alerts generated by onboard diagnostic systems, including timestamps, alarm periods (start to end), and descriptions.
- **Stoppages**: Information on each turbine's stoppages, such as turbine ID, cause of stoppage, and stoppage start/end times.
- **Work Orders and Contextual Information**: Maintenance records for each turbine, covering order ID, creation date, functional location, and other attributes.

- **Notifications**: Additional maintenance details alongside work orders, including malfunction timestamps, descriptions, and details on parts repaired or replaced.

DataFrame Name	Required Columns	Description
`scada`	`COD_ELEMENT`	Unique code identifier for a Turbine
	`TIMESTAMP`	Time index
`pidata`	`COD_ELEMENT`	Unique code identifier for a Turbine
	`time`	Time index
`vibrations`	`COD_ELEMENT`	Unique code identifier for a Turbine
	`timestamp`	Time index
`turbines`	`COD_ELEMENT`	Unique code identifier for a Turbine
`alarms`	`COD_ELEMENT`	Unique code identifier for a Turbine
	`DAT_START`	Start date for an Alarm
	`DAT_END`	End date for an Alarm
	`IND_DURATION`	Duration of Alarm
`stoppages`	`COD_ELEMENT`	Unique code identifier for a Turbine
	`DAT_START`	Start date of Stoppage
	`DAT_END`	End date of Stoppage
	`IND_DURATION`	Duration of Stoppage
	`IND_LOST_GEN`	Generation lost due to Stoppage
`work_orders`	`COD_ELEMENT`	Unique code identifier for a Turbine
	`COD_ORDER`	Unique code identifier for a Work Order
	`DAT_BASIC_START`	Start date of Work Order
	`DAT_BASIC_END`	End date of Work Order
`notifications`	`COD_ORDER`	Unique code identifier for a Work Order
	`DAT_POSTING`	Posting date of Notification
	`DAT_MALF_END`	End date of malfunction
	`IND_BREAKDOWN_DUR`	Duration of breakdown

Table 2.3: Input Data Rules. Here we list the columns required for each *DataFrame* in order for Zephyr to properly generate an *EntitySet*. Note that the column names of the Required Columns are the default names and can be modified by the user.

Data Input Rules Zephyr expects a `pandas` *DataFrame* corresponding to one or more of either `SCADA`, `PI`, and/or `Vibrations` data, and then one of each of the categories of operations data. For each of the *DataFrames*, Zephyr requires specific columns. These required columns define the relationships between tables, allowing us to construct our *EntitySet* structure accurately. Table 2.3 shows the required columns for each *DataFrame*. Code Listing 2.1 shows how to use Zephyr to create an *EntitySet*.

Now that we have created an *EntitySet*, every downstream step can take advantage of the data and relationships contained in that structure.

```
import os
import pandas as pd
from zephyr_ml import create_scada_entityset

data_path = 'notebooks/data'

data = {
    'turbines': pd.read_csv(os.path.join(data_path, 'turbines.csv'))
        ,
    'alarms': pd.read_csv(os.path.join(data_path, 'alarms.csv')),
    'work_orders': pd.read_csv(os.path.join(data_path, 'work_orders.
        csv')),
    'stoppages': pd.read_csv(os.path.join(data_path, 'notifications.
        csv')),
    'notifications': pd.read_csv(os.path.join(data_path, '
        notifications.csv')),
    'scada': pd.read_csv(os.path.join(data_path, 'scada.csv'))
}

brake_pad_es = create_scada_entityset(data)
```

Code Listing 2.1: Creating an *EntitySet*. We first define the `data_path` as the folder *notebooks/data*, where all CSV files are stored. Each file is named according to its data type (e.g., alarms.csv for alarms data). Using `pandas' read_csv` function and `os.path.join`, we load each CSV file into a *DataFrame* and store them in a dictionary called `data`. This dictionary maps each data type to its corresponding *DataFrame*. Finally, we pass the data dictionary into the `create_scada_entityset` function, which generates our *EntitySet*, tailored to the `SCADA` signal data.

2.1.3 Labeling & Generating *LabelTimes*

In machine learning, the prediction problem centers around identifying the target label, or outcome, that the model will learn to predict. Zephyr's labeling functions and `DataLabeler` class are used to define the prediction problem.

Defining the Labeling Function The first step is to create a labeling function, which applies criteria to identify specific events or outcomes in the dataset. This function enables a label search, which determines *LabelTimes*: specific points or intervals in the dataset where the target label is assigned. In our case, the target label is whether a brake pad issue occurred during a turbine stoppage.

Zephyr provides the ability to create custom labeling functions based on domain knowledge. Alternatively, it offers several predefined common labeling functions, which can be accessed as shown in Code Listing 2.2.

```
1 from zephyr_ml import labeling
2
3 labeling.get_labeling_functions()
```

Code Listing 2.2: Getting Zephyr's Labeling Functions

Brake Pad Presence Labeling Function For our prediction problem, we will use the predefined `brake_pad_presence` labeling function, designed to detect whether a brake pad issue was the cause of a turbine stoppage. This is ideal, as our objective is to predict whether a turbine will experience a future stoppage due to a brake pad issue.

The `brake_pad_presence` labeling function uses the following features:

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

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

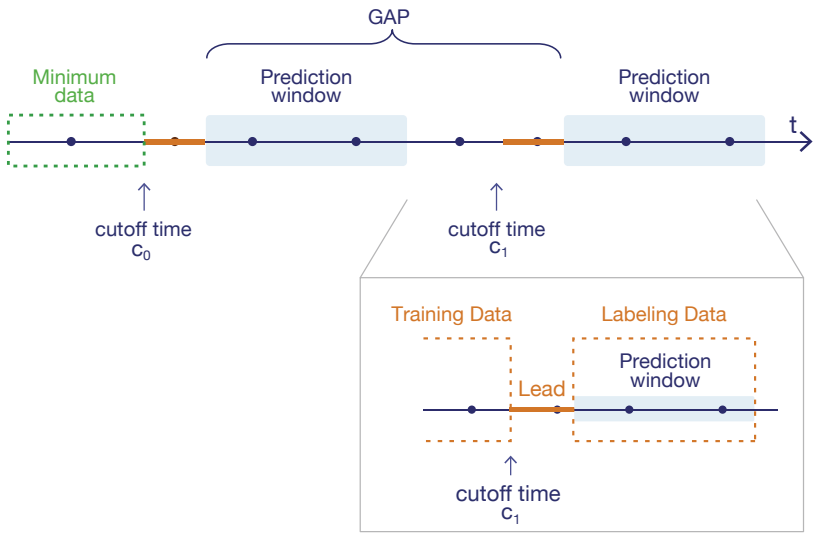


Figure 2-2: Generating *LabelTimes* Parameters [3].The process of generating *LabelTimes* from time-series data is governed by a set of parameters: `minimum data`, `cutoff time`, `prediction window`, and `gap`. The `prediction window` is the duration of data considered for generating each label and begins at a specific time, namely the `cutoff time`. The `minimum data` parameter specifies the initial portion of the dataset that is excluded from the first `prediction window`. The `gap` denotes the temporal interval between successive `cutoff times`. Both the `prediction window` and `gap` are applied uniformly across all data slices and remain constant throughout.

DataLabeler The `DataLabeler` class is initialized with a labeling function, and its `generate_label_times` method uses the labeling function along with additional parameters to slice the data within an *EntitySet*, and to generate *LabelTimes* by applying criteria to identify specific events or outcomes in the data. Figure 2-2 shows an overview of the different parameters used to generate *LabelTimes*. Code Listing 2.3 shows how to perform the labeling and generating *LabelTimes* step for our brake pad problem.

```
from zephyr_ml import labeling, DataLabeler

brake_pad_prescence_labeling_fn = labeling.labeling_functions.
    brake_pad_presence

data_labeler = DataLabeler(brake_pad_presence_labeling_fn)
label_times, metadata = data_labeler.generate_label_times(
    brake_pad_es)
```

Code Listing 2.3: Generate *LabelTimes* with predefined labeling functions. We use the predefined *brake_pad_presence* labeling function to create a DataLabeler instance. We then generate our *LabelTimes* from our *EntitySet*, *brake_pad_es*, by passing it into the *generate_label_times* function. Figure 2-3 shows an example of generated *LabelTimes*.

Visualizing the distribution of our data is crucial for performing sanity checks and providing clarity when exploring a machine learning problem. Code Listing 2.4 shows how we would plot our generated *LabelTimes*.

```
import composeml as cp
cp.label_times.plots.LabelPlots(label_times).distribution();

plt.show()
```

Code Listing 2.4: Visualizing Data Distribution. Zephyr uses `composeml` under the hood to generate *LabelTimes*. We use `composeml`'s `LabelPlots` class to plot our generated *LabelTimes*, `label_times`. Figure 2-4 shows an example visualization.

	COD_ELEMENT	time	label
0	15430.0	2020-01-10 11:28:00	False
1	15430.0	2020-01-30 11:28:00	False
2	15430.0	2020-02-19 11:28:00	False
3	15430.0	2020-03-10 11:28:00	False
4	15430.0	2020-03-30 11:28:00	False
...	...	...	...
2436	15915.0	2021-08-30 22:47:00	False
2437	15915.0	2021-09-19 22:47:00	False
2438	15915.0	2021-10-09 22:47:00	False
2439	15915.0	2021-10-29 22:47:00	False
2440	15915.0	2021-11-18 22:47:00	False

Figure 2-3: Example *LabelTimes*. In this case, we have generated 70 *LabelTimes*. COD_ELEMENT corresponds to the unique wind turbine code. Timestamp corresponds to the specific time associated with each label. Label corresponds to the outcome of our labeling function for that specific *label time*.

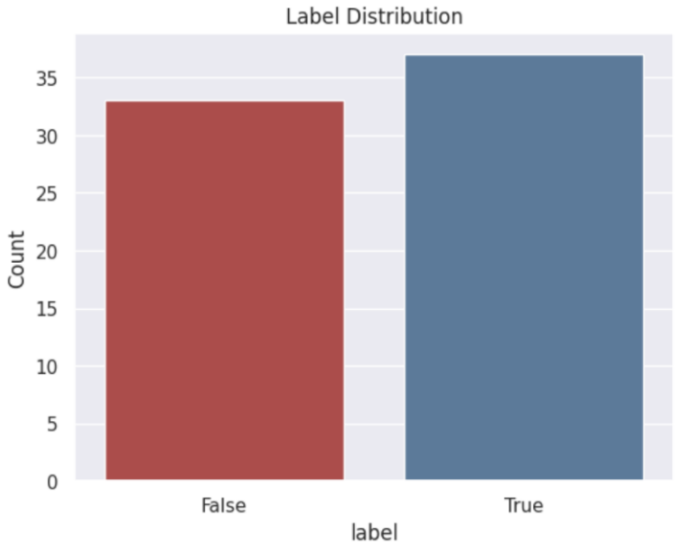


Figure 2-4: Example *LabelTimes* label distribution. True indicates a stoppage was caused by a brake pad issue and False otherwise

2.1.4 Feature Engineering

What are features? Features are the pieces of variable information the machine learning model uses to make predictions. Feature generation is the process of creating new features from the existing ones in a dataset to improve model performance and accuracy.

Creating a feature manually Let's try to create a feature for our current problem. Looking again at our generated *LabelTimes*, each *label time* has the following fields: *COD_ELEMENT*, time, and label.

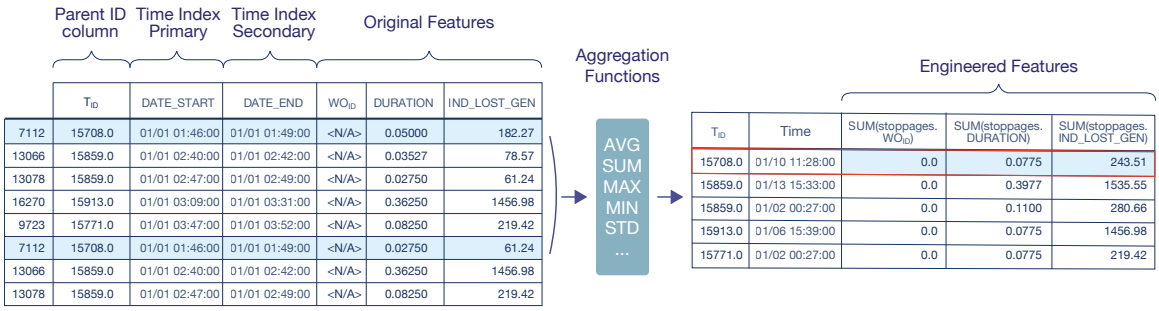
One example of a feature we could create for a specific *label time* would be the number of alarms that occurred for the turbine corresponding to the specific `COD_ELEMENT` at the specific time. Let's name this feature `alarm_count`. To calculate `alarm_count` manually for a specific *label time*, we would have to filter our `alarms` table for such and then count the number of entries that remain.

From the input data rules table above, we see that each entry in the alarms table must contain the following fields: `COD_ELEMENT`, `DAT_START`, `DAT_END`, `IND_DURATION`, and `IND_LOST_GEN`. How does this help? Well, we can now filter our `alarms` table and only keep the entries that meet two criteria: 1) they correspond to the `COD_ELEMENT` we are interested in, and 2) the entry occurred at a time before our *label time*'s time (which can be determined using the entry's time index, `DAT_START`). Finally, we can count how many entries remain to get the `alarm_count` value for our *label time*.

As you can see, it can be very time consuming to do this manually – so far, we have generated just one feature value for one *label time*, and this tutorial includes 70 *LabelTimes*, with many more additional features we would want to generate for each. Fortunately, we can use a framework called `Featuretools` [5] to help automate much of this process.

Featuretools' Deep Feature Synthesis

`Featuretools` is an automated feature generation framework that leverages the *EntitySet* data structure to generate new features from the relational structure of the



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

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

Figure 2-5: Overview of Feature Engineering Step [3]. The first table, representing the `stoppages` table, highlights the parent ID column used as an identifier, the time index columns, and the original features. The aggregation functions are then applied on the original features in order to obtain the second table, which includes the ID column, the time and the engineered features -i.e. the features obtained from the originals after aggregation functions are applied. Note that here we write `DAT_START` as `DATE_START` and `DAT_END` as `DATE_END` for clarity.

data. `Featuretools' Deep Feature Synthesis` (`dfs`) automates feature engineering for relational and temporal data by recursively applying data manipulation and aggregation primitives to transform or aggregate columns across relationships [5]. An overview of the automated feature engineering step is shown in Figure 2-5. Essentially, `dfs` streamlines the process of applying statistical functions on aggregated data. We show an example `dfs` call in Code Listing 2.5.

```
import featuretools as ft

brake_pad_es.add_interesting_values(dataframe_name = "turbines")

feature_matrix, features = ft.dfs(
        entityset = brake_pad_es,
        target_dataframe_name = "turbines",
        cutoff_time = label_times,
        cutoff_time_in_index = True,
        where_primitives = ["count", "sum", "max"],
        agg_primitives = ["count", "sum", "max"],
        verbose = True
```

```
)
```

Code Listing 2.5: Example usage of `dfs`. We can use `Featuretools`' `dfs` with our *EntitySet*, `brake_pad_es` and our generated *LabelTimes*, `label_times` as the `cutoff_time`), representing the specific points in time for which features are calculated, to create new features from our current data. The main feature generation process relies on primitives, which are functions that create features by performing computations on raw data. `agg_primitives` specify the aggregation primitives for generating features along columns and `where_primitives` specify the primitives for generating features along columns that are conditioned on the interesting values we specified prior. In our code block, we have already added `turbines`, our turbines *DataFrame*, as an interesting value. This means that all the categorical columns in our `turbines` dataset will be interesting values.

Using `Featuretools` allows us to enrich the dataset and provide our model with more detailed information for accurate predictions. However, there are a few small technical issues that must be addressed in order to fully integrate this step with the current state of Zephyr, which we address in Code Listing 2.6. A snapshot of our generated feature matrix is shown in Figure 2-6.

```
import pandas as pd

# Drop label and all ID columns
feature_matrix = feature_matrix.drop(labels = feature_matrix.filter(
    regex= "_ID|label").columns.values, axis = 1)

# Find and cast all COUNT columns to be np.int64
count_cols = feature_matrix.filter(like= "COUNT").columns
feature_matrix[count_cols] = feature_matrix[count_cols].apply(lambda
     x: x.astype(np.int64))

# Find and one hot encode string columns
string_cols = feature_matrix.select_dtypes(include= "category").
    columns
```

COD_ELEMENT	time	COUNT(alarms)	MAX(alarms.IND_DURATION)	SUM(alarms.IND_DURATION)	COUNT(stoppages)	MAX(stoppages.COD_WO)
15430.0	2020-01-10 11:28:00	0	NaN	0.0	1	NaN
15431.0	2020-01-13 15:33:00	0	NaN	0.0	1	NaN
15432.0	2020-01-02 00:27:00	0	NaN	0.0	1	NaN
15449.0	2020-01-06 15:39:00	0	NaN	0.0	1	NaN
15450.0	2020-01-04 19:28:00	0	NaN	0.0	1	NaN

Figure 2-6: Example Feature Matrix (Truncated). Each row corresponds to a `COD_ELEMENT` (turbine identifier) at a specific timestamp. For example, the first row represents `COD_ELEMENT` 15430 at 2020-01-10 11:28:00. We can see that at this time, there were 0 alarms (`COUNT(alarms)`) and 1 stoppage (`COUNT(stoppages)`). The feature matrix includes many additional features, providing valuable insights for our prediction problem.

```
feature_matrix = pd.get_dummies(feature_matrix, columns =
    string_cols)
```

Code Listing 2.6: Data Cleaning. Before we can continue using our generated feature matrix, we must drop unnecessary columns, cast integer columns, and one-hot encode categorical columns.

2.1.5 Modeling

Splitting Data into Train and Test Usually, training ML models involves splitting the data into a training set and a testing set, which we show how to do using `sklearn` in Code Listing 2.7. The training set is used to fit the model, while the testing set assesses how well the model generalizes to unseen data, helping us avoid overfitting.

```
from sklearn.model_selection import train_test_split

y = [0 if x == False else 1 for x in label_times["label"].values]

```

```
X_train, X_test, y_train, y_test = train_test_split(feature_matrix,
    y, test_size = 0.3, shuffle = True, stratify = y, random_state =
    15)
```

Code Listing 2.7: We use `sklearn`'s `train_test_split` function to split our feature matrix, `feature_matrix`. We start by creating our label vector `y` using the `label` column from our generated *LabelTimes*, `label_times`. In our example, `test_size` = 0.3 indicates that we want 30% of the dataset to be included in the test split. We set `shuffle` to "True" to indicate that we want the order of our data to be shuffled, and we set `stratify` to `y` to indicate that `y` contains our class labels and that we want to split data such that the label distribution is similar in both the train and test splits. Stratifying allows us to get training data that is more representative of the entire dataset. Finally, setting a `random_state` ensures that we will get the same results each time we run our code.

Training and Testing Zephyr provides a Zephyr class instance that wraps over an `MLBlocks` `MLPipeline`, and can fit to our train data and predict and evaluate our test data, as shown in Code Listing 2.8.

MLBlocks

`MLBlocks` is a framework for seamlessly combining any possible set of machine learning tools developed in Python – whether they are custom developments or belong to third party libraries – and build pipelines out of them that can be fitted and then used to make predictions. This is achieved by providing a simple and intuitive annotation language that allows the user to specify how to integrate with each tool (known as primitives) in order to provide a common, uniform interface for each of them [7].

The `MLBlocks` framework follows the primitive-pipeline design pattern. Users can freely combine, reuse, and replace primitives without repeatedly integrating third-party tools or writing extensive glue code. Meanwhile, pipelines provide a concise, flexible approach to defining reusable workflows that do not need to be rewritten when any individual component is revised. `MLBlocks` is a code-efficient framework

that improves transparency, lowers error potential, and encourages constructive development practices with thorough unit testing and documentation [2].

By default, Zephyr uses an `MLBlocks` `MLPipeline` comprised of an `XGBoost` classifier and a custom postprocessing primitive, `FindThreshold`. The `XGBClassifier` primitive is a model that returns the probability of each class, and the `FindThreshold` primitive creates binary labels from the output of the `XGBClassifier` model by choosing a threshold that produces the best metric value.

```
from zephyr_ml import Zephyr

zephyr = Zephyr(pipeline = "xgb_classifier")

zephyr.fit(X_train, y_train)

y_pred = zephyr.predict(X_test)
```

Code Listing 2.8: Training and Testing using Zephyr. We initialize our Zephyr instance and then fit to our training data, `X_train` and `y_train`. With our trained model, we can use our test data, `X_test` and `y_test`, to predict labels from new data, or evaluate our model on different metrics.

2.1.6 Evaluation

Evaluating our model provides insight into its performance on unseen data, helping us to understand its ability to generalize to new situations. We can use Zephyr's `evaluate` method as shown in Code Listing 2.9 to compute the following metric scores: *accuracy*, *recall*, *precision*, and *F1*.

```
scores = zephyr.evaluate(X_test, y_test)
```

Code Listing 2.9: Zephyr's Evaluate Method. We evaluate our model by passing in our test data, *X_test* and *y_test*, to the `evaluate` method of our Zephyr instance.

Let's briefly define these metrics and their significance:

accuracy	0.619048
f1	0.714286
recall	0.909091
precision	0.588235

Figure 2-7: Example Evaluation Output

- **Accuracy**: The proportion of all classifications that the model made correctly. In our case, it represents the fraction of data points correctly classified as having or not having brake pads involved in turbine failure. Accuracy offers a general view of model quality, which is especially useful for balanced datasets.

- **Recall (True Postive Rate)**: The proportion of actual positives that were correctly classified as positives. In our context, this is the fraction of data points with brake pad involvement that were accurately classified as such. For imbalanced datasets, where actual positives may be sparse, recall can sometimes be less informative.

- **Precision**: The proportion of all positive classifications that are truly positive. In our example, it's the fraction of data points classified as having brake pad involvement that actually did. Like recall, precision can be limited in meaning for imbalanced datasets.

- **F1 Score**: The harmonic mean of precision and recall. It assesses a model's ability to maintain both high precision and high recall. While it offers a balanced view, the F1 score alone may not fully capture performance in heavily imbalanced datasets.

More precision involves a harsher critic (classifier) that doubts even the actual positive samples from the dataset, thus reducing the recall score. On the other side, more recall involves a lax critic that allows any sample that resembles a positive class to pass, thus classifying border-case negative samples as "positive" and reducing the precision.

A hypothetical perfect model would have a score of 1.0 for all metrics described. Looking at the evaluation metric scores that Zephyr calculated in Figure 4-3, it seems our model scored pretty well on recall, but was a bit lackluster in other metrics. To

understand why, we need to perform further evaluation and analysis, which can reveal potential areas for tuning or improvement.

Zephyr does not natively support some common postprocessing directives, but we can extract the underlying MLPrimitives to perform further evaluation and analysis, as shown in Code Listing 2.10.

```
xgb_classifier = zephyr._mlpipeline.blocks['xgboost.XGBClassifier#1'
    ].instance
```

Code Listing 2.10: Extracting XGBClassifier from Zephyr. Here we get the XGBClassifier instance from our Zephyr instance's, *zephyr*, MLPipeline.

Extracting the main classifier model allows us to perform additional analysis, such as visualizing metric scores across different classification thresholds or retrieving prediction probabilities.

Confusion Matrix We can also plot a confusion matrix for analysis. A *confusion matrix* shows how many samples from each label were predicted correctly, allowing us to see where the model fails and where it succeeds. We show how one may plot a confusion matrix in Code Listing 2.11.

```
from sklearn import metrics
import seaborn as sns

conf_matrix = metrics.confusion_matrix(y_true = y_test, y_pred =
    y_pred)
ax = sns.heatmap(conf_matrix, annot = True, cmap = "Blues")

ax.set_title("Confusion Matrix \n", size = 18)
ax.set_xlabel("\nPredictedValues", size = 18)
ax.set_ylabel("Actual Values", size = 18)

# Set ticket labels along the x and y axis
ax.xaxis.set_ticklabels(['False', 'True'], size = 18)
ax.yaxis.set_ticklabels(['False', 'True'], size = 18)
```

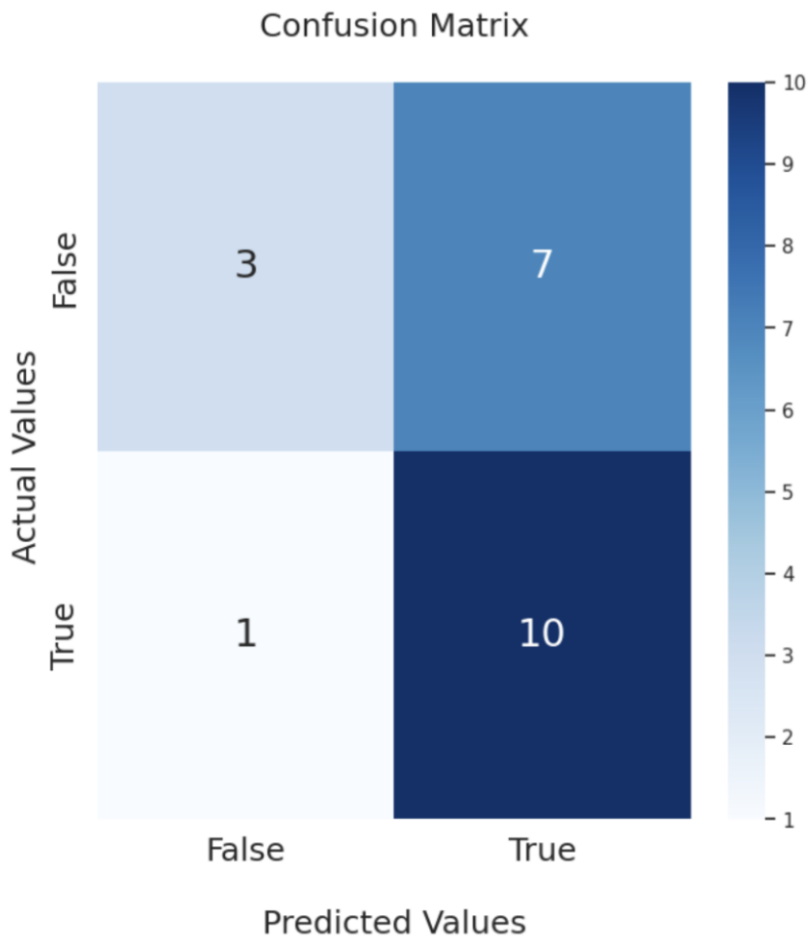


Figure 2-8: Example Confusion Matrix

```
14
15 plt.show()
```

Code Listing 2.11: Plotting a Confusion Matrix. We create our confusion matrix using `sklearn.metrics`' `confusion_matrix` function and pass in our test labels, `y_test` and predictions `y_pred`. We also use `seaborn` to add a heatmap to our confusion matrix using the `heatmap` function. Finally, we add labels to make our confusion matrix easier to read using the respective `matplotlib` methods. An example confusion matrix plot is shown in Figure 2-8.

AUC-ROC Curve Another insightful tool for evaluating and visualizing model performance is the plotting of a *Receiver Operating Characteristic Curve* (ROC curve).

The ROC curve is a plot of the True Positive Rate (TPR) against the False Positive Rate (FPR) at different classification thresholds (remember that as you vary the classification thresholds, you get different TPR and FPR).

As a reminder, the True Positive Rate is the proportion of actual positive cases that were correctly identified or classified as positive. The False Positive Rate is the proportion of all actual negatives that were classified incorrectly as positives.

The ROC curve is a probability curve, and AUC is the area under the ROC curve. It is a metric that measures how well the model can distinguish between two classes, where the higher the AUC, the better the performance. The AUC-ROC is useful because it gives us a comprehensive view of the model's performance across all possible thresholds, and its trade-off between true positives and false positives.

We show how to plot an ROC curve and calculate the AUC score in Code Listing 2.12.

```
from sklearn import metrics
import matplotlib.pyplot as plt

y_pred_prob = xgb_classifier.predict_proba(X_test)[:, 1]
fpr, tpr, _ = metrics.roc_curve(y_test, y_pred_prob)

ns_probs = [0 for _ in range(len(y_test))]
ns_fpr, ns_tpr, _ = metrics.roc_curve(y_test, ns_probs)

# Create ROC curve
fig, ax =  plt.subplots(1, 1, figsize = (5,5))

# Calculate AUC
auc = metrics.roc_auc_score(y_test, y_pred_prob)
print('AUC: %.3f' %auc)
plt.plot(fpr, tpr, "ro")
plt.plot(fpr, tpr)
plt.plot(ns_fpr, ns_tpr, linestyle = "--", color = "green")

```

```
20 plt.ylabel("True Positive Rate")
21 plt.xlabel("False Positive Rate")
22 plt.title("AUC: %.3f" % auc)
23
24 plt.show()
```

Code Listing 2.12: Plotting and Calculating AUC-ROC. We use `sklearn.metrics`' `roc_curve` function to determine our false positive and true positive rates at each classification threshold by passing in our test labels and our prediction probabilities for each label. We also create an array of 0 probabilities to generate false positive rates and true positive rates that create an ROC curve with a slope of 1. This corresponds to a naive model, which we use as a visual baseline. Finally, we can calculate the AUC score using `sklearn.metrics`' `roc_auc_score` function, and plot our ROC curves using `matplotlib.pyplot`'s `plot`. An example plot is shown in Figure 2-9.

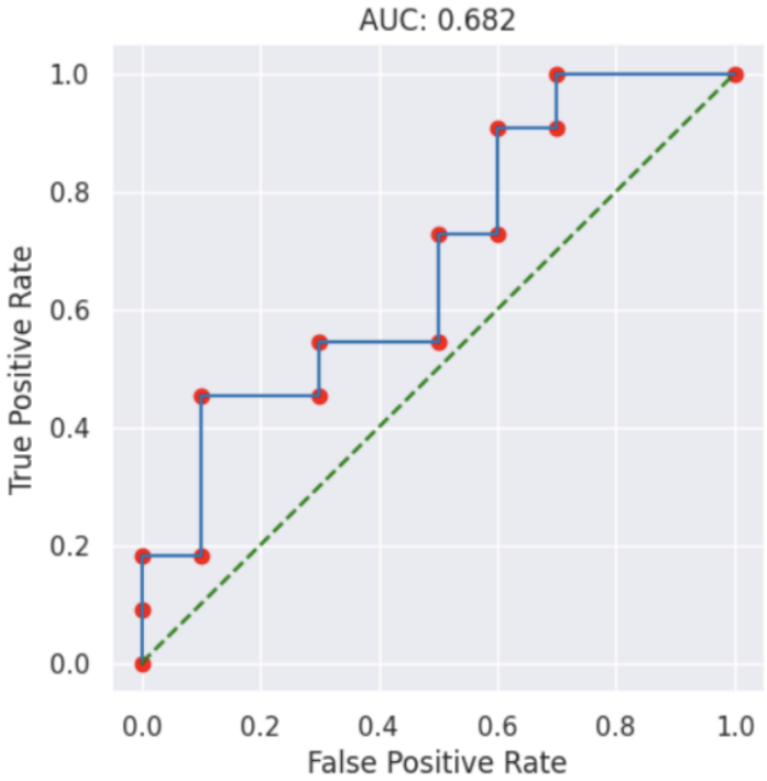


Figure 2-9: Example AUC-ROC plot

Chapter 3

Implementation

3.1 Objective

As shown in the case study, Zephyr enables users to streamline model building for predictive maintenance. However, much of the current tooling is fragmented across different functions and classes, and some crucial steps along the model building process required external tooling. Our goal is to maintain all of Zephyr's current functionality while creating a more consolidated and complete experience for the user.

Our development process for the new Zephyr framework spanned several stages, each stage motivating the next. We started with the overarching issue of fragmentation, and solved this by creating a Zephyr API that encapsulates all the necessary functionalities our users need, mostly shown in our brake pad tutorial.

Implementing a single API for an entire predictive modeling workflow necessitates some form of state management on our end, to allow for consistent behavior across API calls. When done correctly, providing this state management allows for a much simpler user experience. Users can focus less on where their data is located and which step requires what data, and more what the next step will be.

However, the predictive modeling workflow also involves the reiteration of previous of steps. Reiteration, coupled with a single API and state management, is a recipe for user confusion and "pipeline jungles." We want our API to be flexible and allow for reiteration, but we also want to prevent user confusion. Our solution to this comes

in the form of warning messages that are clear and explicit enough to keep users on track, coupled with careful exception handling to prevent any inconsistency within our state management.

Finally, our case study revealed that much of the relevant evaluation and analysis on our end model had to be done with external tooling. In response, we improved our evaluation step, making it broader and more flexible. An overview of the methods in Zephyr that support each step of the predictive modeling workflow is shown in 3.1. This chapter will go into depth about the motivations and implementation behind the overall structure of the new Zephyr API and each individual method.

3.2 Zephyr API

The main goal of Zephyr is to assist in the generation and solving of machine learning problems for wind farm operations data. The Zephyr API upholds this, and the methods exposed encapsulate the entire predictive modeling workflow, allowing users to go from a raw dataset to a fitted model. Because the steps involved in problem generation follow an intrinsic order, the Zephyr API provides additional methods so that users can solve a step outside of Zephyr and then resume within Zephyr, providing more flexibility.

The API is designed to encapsulate the entire general predictive modeling workflow. With that in mind, coupled with the natural ordering of each step, as well as the iterative nature of predictive modeling, it made sense to provide state management to the user, and help handle intermediate output from step to step. State management raises some concerns regarding the consistency of data the API stores and returns, but we are able to address these concerns also with the same idea – that there is a natural ordering of steps. We describe our solution to the consistency concerns, the Guide System, later, in Section 3.3.

Step	Key Method	Set Method	Get Method
Data Preprocessing	`generate_entityset`	`set_entityset`	`get_entityset`
Labeling	`generate_label_times`	`set_label_times`	`get_label_times`
Feature Engineering	`generate_feature_matrix`	`set_feature_matrix`	`get_feature_matrix`
Splitting Data	`generate_train_test_split`	`set_train_test_split`	`get_train_test_split`
Modeling	`fit_pipeline`	`set_fitted_pipeline`	`get_fitted_pipeline`
Prediction	`predict`		
Evaluation	`evaluate`		

Table 3.1: Zephyr's Predictive Modeling Workflow Methods

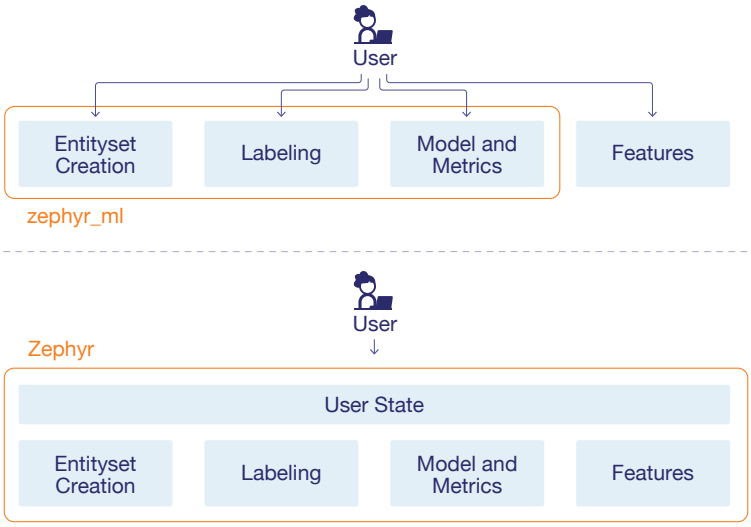


Figure 3-1: Previous (Top) vs New (Bottom) Zephyr. The previous Zephyr did not encapsulate all stages of the predictive modeling workflow and required users to import standalone functions from different `zephyr_ml` modules. The new Zephyr API encapsulates all the stages of the predictive modeling workflow, and manages user state as they go from step to step.

3.2.1 Class Structure

We choose to implement the Zephyr API with a single class to reduce fragmentation and simplify the state management for the user. With state management and the dependence of data from step to step, our methods contain careful error handling and input validation to ensure that each step can be performed correctly. However, managing a user's state given the intrinsic ordering of the steps involved in problem generation reintroduces the pipeline jungle issue. An overview of the changes we make to Zephyr is shown in Figure 3-1.

3.2.2 Key Methods

In this section, we provide an in-depth description of the key methods the Zephyr API exposes, their functionality, and their implementation details.

generate_entityset

This method allows users to generate an *EntitySet* simply by passing in `data_paths` (a dictionary mapping entity names to the `pandas` *DataFrame* for that entity), and `es_type` (the type of signal data the *EntitySet* will represent). Zephyr's supported *EntitySet* types are detailed in Table 3.2. Users can also optionally pass in `new_kwargs_mapping` (for more customized *EntitySets*), and signal processing arguments (to perform signal processing on the generated *EntitySet*). Our signal processing implementation is described in later in section 3.2.3.

Zephyr previously supported individual *EntitySet* creation functions for each signal data type, which was very fragmented and redundant. Much of the difference in *EntitySet* generation for different signal types lies in the validation of the raw data, whether or not we could convert the user's `data_paths` into an *EntitySet*for that specific signal type. Therefore, the functions calls internally contain separate validation functions for each signal type, resulting in less code and a simpler user interface. Finally, the *EntitySet* is stored internally as a class attribute for later steps.

Type	**Data Description**
`SCADA`	Signal data from the Original Equipment Manufacturer Supervisory Control And Data Acquisition (OEM-SCADA) system, a signal data source. Signal data is time series data that originates from wind turbine assets and their auxiliary systems.
`PI`	Signal data from the operator's historical Plant Information (PI) system.
`Vibrations`	Vibration data collected on planetary gearboxes in turbines.

Table 3.2: List of Zephyr's *EntitySet* types.

generate_label_times

This method uses our *EntitySet* and slices the data into a *DataFrame* containing rows referred to as *LabelTimes*. *LabelTimes* are generated based on the chosen `labeling_function` and specific parameters.

Name	Function Description
`brake_pad_presence`	Determines if brake pad is present in stoppages.
`converter_replacement_presence`	Calculates any converter replacement presence.
`gearbox_replace_presence`	Determines if gearbox replacement/exchange is present in stoppages
`total_power_loss`	Calculates the total power loss over the data slice

Table 3.3: Zephyr Labeling Functions

Zephyr previously provided a `DataLabeler` class, which was instantiated with a labeling function and provided the `generate_label_times` function as its sole method. Because choosing a labeling function and performing the *LabelTimes* generation are tightly coupled, we decided to reduce this class into a single method in order to make things simpler for users. Finally, the *LabelTimes* are stored internally as a class attribute for later steps.

The user can either define their own labeling function or choose from one of our predefined functions, listed in Table 3.3. The user can retrieve all predefined functions by calling `GET_LABELING_FUNCTIONS`, one of Zephyr's static help methods. Our static help methods are described later, in section 3.2.3. If using one of our predefined functions, all the user has to do is provide the name of the predefined function via the `labeling_fn` argument.

generate_feature_matrix

This method is implemented as a wrapper over `Featuretools dfs` with some additional functionalities, and is performed on a deep copy of the instance's *EntitySet*. These additional functionalities may modify the instance's *EntitySet*, creating changes inconsistent with the results of the previous steps that rely on the instance's *EntitySet*, so we make a copy of the *EntitySet* at the beginning of this step. Note that we do not store this updated *EntitySet* as an attribute, and simply return it to the user at the end of this method, since this copy was only updated for the purposes of feature generation.

One modification to the *EntitySet* occurs if the user passes in any signal processing arguments, which results in signal processing being performed on the *EntitySet*. The signal processing implementation is the same as in the `generate_entityset` step. Again, our signal processing implementation and related decisions are discussed in section 3.2.3.

After any signal processing, the `dfs` call conveniently defaults to our *EntitySet* copy and the instance's *LabelTimes*. This method also supports adding interesting features to the *EntitySet* copy prior to the `dfs`call, which is used to generate `where` features. After the `dfs` call completes, we clean the feature matrix, and return the cleaned feature matrix, generated features, and the *EntitySet* copy to the user. The `feature matrix` and `features` are stored internally as class attributes for later steps.

Previously, if a user wanted to perform any sort of feature generation, this would have to happen outside Zephyr. Because we expect feature generation to be a very common step for users, this method is meant to seamlessly integrate `Featuretools dfs`into the Zephyr class. Of course, users still have the flexibility to incorporate their own external feature generation tools using the relevant `set` and `set` methods, which we also discuss in section 3.2.3.

generate_train_test_split

This method is implemented as a wrapper over `sklearn.model_selection.train_test_split`, which splits arrays or matrices into random train and test subsets. This method automatically passes in the current class instance's generated feature matrix and labels to be split, and stores them as attributes before returning to the user.

Performing a train test split on a dataset is extremely common in a machine learning workflow, and `sklearn`'s function is a popular choice. Thus, instead of having user's perform their train test split outside of Zephyr, we chose to integrate it, fully supporting the original function's parameters.

fit_pipeline

This method allows the user to select an `MLPipeline` and fit it to the data they have been working with. Specifically, it passes in the class instance's training data into the `MLPipeline`'s `fit` method. Zephyr provides a default modeling pipeline involving `xgboost's XGBClassifer` and a custom `MLBlock`, `FindThreshold`. We discuss Zephyr's default `MLPipeline` in a later section.

In the original version of the Zephyr class, a Zephyr class instance would be initialized with an `MLPipeline`, and one of the main methods would be `fit`, which is essentially what this current method is aiming to provide. However, the instantiation of the class and invocation of the method have been merged into this one method, with the added user convenience of automatically passing in the previously generated training data from within the instance. We choose to merge these two actions, since they are tightly coupled – all the data has been gathered and fitting the model is the natural next step after choosing one. Users with an already fitted pipeline can call the relevant set method, `set_fitted_pipeline`.

predict

This method allows the user to make new predictions using their fitted model. Specifically, it passes the class instance's `test data` into the `MLPipeline`'s `predict` method. This method is very similar to the original version, but with the added user convenience of automatically passing in the previously generated test data from within the data.

evaluate

This method allows the user to evaluate their fitted models. There are many metrics a user might want to use to evaluate their model, and we want to provide support for as many of them as possible. To start, we first identified the most primitive blocks a user can use for evaluation by looking at common evaluation metrics.

At this point, the Zephyr class instance must contain a `fitted pipeline` and

Name	Description	Arguments
sklearn.metrics. accuracy_score	Compute the accuracy.	y_true y_pred
sklearn.metrics. precision_score	Compute the precision.	y_true y_pred
sklearn.metrics. recall_score	Compute the recall	y_true y_pred
sklearn.metrics.f1_score	Compute the F1 score, also known as balanced F-score or F-measure.	y_true y_pred
zephyr_ml.primitives. evaluation.confusion_matrix	Create and plot confusion matrix.	y_true y_pred
zephyr_ml.primitives. evaluation. roc_auc_score_and_curve	Calculate ROC AUC score and plot curve.	y_true y_proba

Table 3.4: Zephyr Evaluation Metrics. Note that we name arguments to be consistent with our XGBClassifier MLPipeline. `y_true` represents the actual labels, `y_pred` represents the predicted labels, and `y_proba` represents the probabilities of the predicted labels.

`test data`, which we can use to produce prediction labels. However, `MLBlocks` supports intermediate outputs and partial execution, which we also believe the user may find helpful for evaluation. When an `MLPipeline` is being executed, a dictionary, referred to as the `Context`, stores the intermediate outputs. A nice attribute of `MLBlocks` is that the outputs in the `Context` dictionary are named based on each `MLPrimitive`'s and the `MLPipeline`'s annotations, which means users have full control over how the `Context` dictionary looks after all steps are run. We refer to the `Context` dictionary after all steps are run as the `Final Context`, and we provide the `Final Context`,to each of the evaluation metrics. Note that the `Final Context` still contains the intermediate outputs that were not overwritten, i.e. given the same variable name from each step. We decided that if a variable name was overwritten from step to step, it would be very complicated to distinguish what is being evaluated, and confusing to the user to try to extract. If a user required intermediate outputs that were being overwritten, we recommend they update their `MLPrimitive` and/or `MLPipeline` annotations to include distinct variable names for intermediate outputs. We retrieve the `Final Context` dictionary by passing in -1 to the `_output` parameter

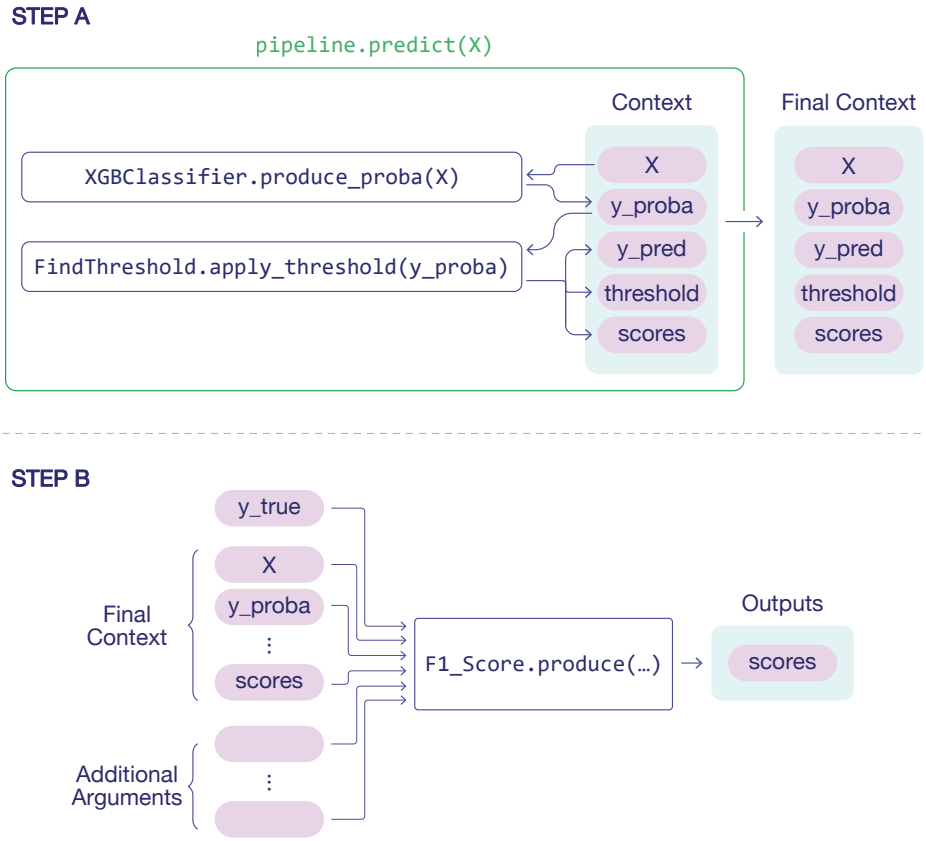


Figure 3-2: Zephyr Evaluation Step. In Step A, we call our fitted pipeline's `predict` method and extract the Final Context library, containing the call's intermediate and final outputs. In Step B, we pass into each evaluation MLPrimitive's, e.g. `F1_Score`'s, produce method the true labels as *y_true*, all the outputs in the Final Context as they are named, and any additional arguments the user has passed in.

to the primitive's `predict` method. Zephyr Evaluation Metrics are listed in 3.4.

The full list of arguments that are passed into each `MLPrimitive`'s `produce` method contains the following: `y_true`, the true labels corresponding to the feature matrix used for evaluation, the `Final Context`, and any additional arguments passed in via the `global_args` and `local_args` parameters of the `evaluate` method. `global_args` are arguments that will be passed in to all evaluation metrics, and `local_args` is a mapping of specific additional args for each evaluation metric. As an additional benefit to the user, `MLBlocks` already provides detailed error messages that indicate if any of the required arguments are not received, which should help users align Zephyr's provided `MLPipeline` and/or their custom pipeline annotations with Zephyr's provided evaluation metrics and/or their custom evaluation metric an-

notations. An overview of the `evaluate` step is shown in Figure 3-2.

To reduce complexity, we recommend users use the same naming convention for their custom metrics' input arguments as Zephyr's evaluation metrics, if they plan to use them. Table 3.4 lists each metric's arguments for reference. Using the same naming convention is not required, and `evaluate` provides two arguments, `global_mapping` and `local_mapping`, that help enable the compatibility of users' custom `MLPipeline`s and evaluation
`MLPrimitive`s. `global_mapping` maps the name of items in the `Final Context` dictionary to what they should be renamed to, while `local_mapping` maps the name of each `MLPrimitive` to a dictionary mapping the primitive's `produce` arguments' names to what they should be renamed to. Our implementation renames the items in `Final Context` using `global_mapping` first, and then performs more fine-grained renaming when passing arguments into each evaluation metric's `produce` method using `local_mapping`.

To get the original name of the arguments, users can refer to Table 3.4, or call the help method `GET_EVALUATION_METRICS` to retrieve the `MLPrimitive` objects and then invoke each primitive's `produce_args` method. Help methods are described in a later section.

The use of `MLBlocks` enables users to formally include their own evaluation metrics and allows for consistent behavior. Previously, Zephyr's `evaluate` method only supported `sklearn` metrics, and only passed in prediction and truth values. Our new implementation fixes this limitation and allows for more types of evaluation within Zephyr.

3.2.3 Additional Methods and Details

In this section, we describe additional methods and implementation details that improve user experience and/or support our key methods.

set and get methods

Each key method except for `predict` and `evaluate` has a corresponding `get` method that allows users to retrieve the result generated from the key method, as shown in Table 3.1. For example, `generate_feature_matrix_and_labels` has the corresponding `get_feature_matrix_and_labels` method. `Get` methods allow users to retrieve any state within their Zephyr class instance. Users may want to re-examine results from prior steps, and since old results are automatically stored for future steps, we are able to provide this convenience to the user for free.

Most of the key methods mentioned in Subsection 3.2.2 have a corresponding `set` method, as shown in Table 3.1. A `set` method allows a user to perform the corresponding step externally and then resume the rest of their predictive modeling workflow in Zephyr. For example, `generate_feature_matrix_and_labels` has the corresponding `set_feature_matrix_and_labels` method. Note that all `set` methods validate the data that is passed in to ensure that all data stored in the user's Zephyr instance allows for consistent behavior in subsequent steps. Thus, although Zephyr fully supports an end-to-end predictive modeling workflow, `set` methods safely enable flexibility and allow users to use whatever subset of Zephyr meets their needs. To add to the flexibility Zephyr provides, users may also couple corresponding `get` and `set` methods to replace or skip the corresponding step in their Zephyr class instance.

Method Name	**Description**
`GET_ENTITYSET_TYPES`	Lists Zephyr *EntitySet* types
`GET_LABELING_FUNCTIONS`	Lists Zephyr labeling functions
`GET_EVALUATION_METRICS`	Lists Zephyr evaluation metrics

Table 3.5: Zephyr Help Methods. Note the capitalized naming convention to distinguish from get methods.

help methods

As mentioned, Zephyr provides support for several *EntitySet* types, and implements several labeling functions and evaluation primitives. As an added convenience to the user, we provide several help methods that allow users to examine what is available

within Zephyr at runtime. Table 3.5 lists the available `help` methods. Each of them returns a dictionary mapping the name of the object to the actual object that can be used as a parameter, as well as its description. This return format allows the user to learn what is available, make a choice, and retrieve what they need for a parameter to a method, all with a single call. The descriptions are the same as shown in Tables 3.2, 3.3 and 3.4.

Primitive	**Type**	**Description**
`crest_factor`	`Aggregation`	Computes and returns the crest factor (ratio of peak to RMS) of the input values.
`kurtosis`	`Aggregation`	Computes and returns the kurtosis of the input values.
`mean`	`Aggregation`	Computes and returns the arithmetic mean of the input values.
`rms`	`Aggregation`	Computes and returns the root mean square (RMS) of the input values.
`skew`	`Aggregation`	Computes and returns the skew of the input values.
`std`	`Aggregation`	Computes and returns the standard deviation of the input values.
`var`	`Aggregation`	Computes and returns the variance of the input values.
`band_mean`	`Aggregation`	Filters between a high and low band and computes the mean value for this specific band.
`identity`	`Transformation`	Returns the input amplitude values as its output.
`power_spectrum`	`Transformation`	Applies an RFFT on the amplitude values and returns the real components.
`frequency_band`	`Transformation`	Filters between a high and low band frequency and return the amplitude values and frequency values.
`fft`	`Transformation`	Applies an FFT on the amplitude values.
`fft_real`	`Transformation`	Applies an FFT on the amplitude values and returns the real components.
`stft`	`Transformation`	Computes and returns the short time Fourier transform.
`stft_real`	`Transformation`	Computes and returns the real part of the short time Fourier transform.

Table 3.6: `SigPro` Primitives.

Signal Processing

When working with a large collection of raw sensor or log signals, transforming those streams into informative, noise-robust summaries is often the most effective way to boost model quality. Effective signal processing distills the data into features that expose domain structure and temporal patterns, giving the downstream ML model more expressive and reliable inputs.

As mentioned, Zephyr provides signal processing support at the `generate_entityset` and `generate_feature_matrix` steps. At these two stages, we automatically perform signal processing on the instance's *EntitySet*. Our implementation relies on the `SigPro` library, a Python-based toolkit that allows subject-matter experts to construct sophisticated pipelines from modular building blocks, or `primitives`. To prevent confusion from `MLBlocks`'s `MLPipelines`, we will refer to these primitives as `SigPro` primitives. `SigPro` primitives transform and aggregate raw time series data into meaningful features, embedding domain knowledge directly into the features and thus improving both the performance and interpretability of downstream predictive models.

Previously, Zephyr provided a function, `process_signals`, that users would use to perform the same signal processing on their own. The function would apply `SigPro` `transformations` and `aggregations` on the specified entity from the given *EntitySet*. This was an attempt to integrate `SigPro` within Zephyr, but ultimately still felt fragmented. One challenge of integrating our signal processing step is that it does not necessarily fit into our intuitive predictive engineering step ordering, because users may want to use it before or after generating *LabelTimes* (or even both). Therefore, we decided that in order to properly integrate it in our workflow, we needed to include it as a subprocess in both aforementioned methods. Users can perform signal processing as a part of `generate_entityset` and `generate_feature_matrix` by passing in the relevant signal processing arguments, detailed in the Appendix. Table 3.6 lists the available `SigPro` primitives.

XGBClassifier and FindThreshold

XGBClassifier Zephyr uses `MLBlocks`' `MLPipeline` class to implement the modeling pipeline. The use of `MLBlocks` allows a lot of flexibility for users, who can easily change and modify primitives as they see fit. We now describe Zephyr's default modeling pipeline, a custom `XGBClassifier` pipeline. Zephyr's default modeling pipeline starts with the `XGBClassifier`model. Gradient-boosted tree ensembles, i.e. `XGBClassifier`, have repeatedly demonstrated competitive performance on tabular, structured data problems. [1] used `XGB` with their prediction framework, `Cardea`, where features are generated automatically with `Featuretools`. They showed that `XGB` was the best classifier in most cases, and that it performed competitively against classical methods with manually generated features in terms of F1 scores. `Cardea`'s setup is very similar to that of Zephyr's feature engineering and modeling steps, just in different domains. The compatibility of `XGB` and `Featuretools` makes setting the `XGBClassifier` as the default model a clear choice. However, the `XGBClassifier` defaults to using a threshold of 0.5, and does not provide any solution to optimize the classifier with respect to its threshold. To fix this, Zephyr does two things. Zephyr's `MLPrimitive` annotation of the `XGBClassifier` has its `produce` method as `predict_proba`. This has the classifier produce raw probability scores instead of labels. Zephyr then appends a custom `MLPrimitive` `FindThreshold`, which takes in the raw probability scores to make predictions on to complete the `MLPipeline`.

FindThreshold To turn raw probability scores into labels, Zephyr appends a custom `FindThreshold` `MLPrimitive` immediately after the `XGBClassifier` `MLPrimitive` in its custom `XGBClassifier` `MLPipeline`. During the `fit` stage of this pipeline, this component sweeps candidate cut-points on the training data and chooses the threshold that best satisfies an evaluation metric, such as maximizing accuracy, precision, recall, or F_1 score. The selected cutoff is stored as a class attribute and used to determine labels at the `predict` stage. Because `FindThreshold`is itself an `MLPrimitive`, developers can reuse it in other custom pipelines or replace it with an alternative search strategy without altering the core model.

3.3 Guide System

We have worked extensively on case studies using Zephyr in `Jupyter Notebooks`, which is an ideal platform as it offers an interactive space for development and analysis without much setup. However, as projects grow in complexity (or, more commonly, reiterate on old code), the linear and sometimes fragmented nature of notebooks can become challenging and confusing. It's easy to lose track of the execution order, and with cells scattered throughout, understanding the overall flow and the dependencies between different code blocks can become quite confusing. This can lead to difficulties in reproducibility and make it harder to revisit and understand the logic. We feel that this is a common headache, especially with less technical users.

To address this, we developed the Guide System, designed to (1) warn users when they enter, or exist in, what we call a *stale* state (i.e. when they stray from the ordering of the steps) and (2) guide users back to an up-to-date state. We also want users to be able to iterate on different choices at various steps and be confident that they are on the right track.

Further, although a user's state may be *stale*, it must not be what we term *inconsistent*. The Guide System includes careful state management of step ordering and the Zephyr methods contain careful error handling to ensure that the user's state is always consistent. Ultimately, the Guide System is tailored to give users a more structured and less confusing experience when working with Zephyr and their predictive modeling workflow.

Step Management

In order to implement the Guide System, we need to encode the ordering of steps within Zephyr. The ordering of steps reflects the intrinsic order of the predictive modeling workflow. Table 3.1 shows each of the steps in order. All corresponding steps, i.e. each row in Table 3.1 are given the same step number.

With this step encoding, we can now define the notion of an `iteration`. An `iteration` refers to the user performing some subset of the predictive modeling work-

flow steps in order, where each step relies on the result of the previous step. Thus, a new `iteration` begins each time the user calls a set method, which sets data that may not necessarily be related to the results from the steps prior. A new `iteration` also begins if a user goes back and redoes some subset of steps using key methods to try and improve their results. We refer to the act of performing a step before the `current step`, the latest step of the current `iteration`, a backwards step. A valid backwards step will begin a new `iteration` and update the `current step`. A valid backwards step refers to the performance of a `key` method at a step before the `current step` where the step prior to the backwards step is not *stale*.

We define a step being in a *stale* state if its result is not a part of the current `iteration`. For example, suppose we are on `iteration` 0, and we have performed steps 0-5 in order with `key` methods. Steps 0-5 are considered part of `iteration` 0. Now suppose we perform a valid backwards step at step 0, which would increment the `iteration` to 1. Steps 1-5 are now considered to be in a *stale* state.

Building off the previous example, suppose we call a `set` method at step 1, which would begin `iteration` 2. Step 0 is still at `iteration` 1, and would be considered *stale*, which is desired. However, suppose we continue with `iteration` 2, and perform key methods at steps 2-5, and then perform a valid backwards step at step 3, beginning `iteration` 3. The result of step 3 must rely on the result of step 2, which means that step 2 should not be considered *stale*. Thus, we introduce the notion of a `start point`, which is the initial step of the latest subset of steps that have been performed in order. Since `set` methods can set data that can be *stale*, or in general, unrelated to the results from the steps prior to the corresponding step, we update the `start point` to be the corresponding step to the `set` method, in addition to incrementing the `iteration`. Therefore, if we perform a valid backwards step, we can update the `iteration` number of all steps from the current *start point* to the backwards step to be the newest `iteration`. With this in place, in our running example, our *start point* should be at step 1, due to the `set` method at step 1. Steps 2 and 3 are now a part of 3, but step 0 is still at `iteration` 1, since it is before our *start point*. Finally, steps 4 and 5 should still be a part of `iteration` 2. `Get` methods are allowed to retrieve

stale data, with a warning, but `key` methods are not allowed to use *stale* data. Note that performing step 0's `key` method will reset the `start point` to 0.

As mentioned, the steps of the predictive modeling workflow require every step to rely on the step before. However, we do not want the user to use *stale* data to perform later steps, so we consider the direct usage of *stale* data via key methods to be *inconsistent* behavior. Continuing with our example, suppose we now attempt to perform step 5 using a `key` method. The `key` method of step 5 must use the result from step 4, but step 4's result is *stale* since it is part of `iteration` 2 when the current `iteration` number is 3! Therefore, performing a `key` method when the previous step has not been performed or is *stale* is incorrect and leads to an *inconsistent* state within the Zephyr instance. Note that we never want a user to be in an *inconsistent* state, which is why we do not allow these methods to be invoked. Our Guide System should prevent these steps from being called, and log an appropriate warning to the user.

There may exist cases where the user may want to re-perform some subset of steps using *stale* data. Although our system prevents users from directly using *stale* data, a user may call the `get` method of a *stale* step, and then call the corresponding `set` method with the *stale* data to begin a new `iteration` with the *stale* data. This is a supported behavior and aligns with our system's staleness and consistency guarantees. Ultimately, we do not really expect users to continuously reuse *stale* data, but rather to make forward progress within Zephyr in the majority of cases. The most common case is meant to be seamless – and even in the extreme case where the user continuously performs external steps using `set` methods each `iteration`, that's only 2 extra method invocations.

Guide Messages

Despite the complexity of our step management process, we hope that the warning messages produced by our Guide System will provide a more straightforward and seamless user experience. These messages should always provide guidance so that forward progress can be made, and also show common alternatives.

`Get` methods will result in a warning if the data being returned is *stale* or does not exist. This warning message will include the corresponding `key` and `set` method that should be re-run to produce up-to-date data. The message will also include the steps that are up to date, which are the steps in between the `start point` and `current step`.

`Set` methods will result in a warning when a `set` method is performed, indicating which steps will be skipped, and that prior steps will be *stale*. If the `set` method is a backwards step, the message will also indicate that the steps after will also be *stale*.

`Key` methods will result in a warning when the previous step is *stale*. The message will include instructions on how to get and set the data of the previous step if they want to continue with *stale* data, the previous step's `key` method to attempt to bring the previous step up-to-date, and the corresponding `set` method if the user already has the data for this step.

`GuideHandler` Class

To decouple our Guide System from Zephyr, we implement our step management in a separate `GuideHandler` class. An overview of how the `GuideHandler` interacts with the user and Zephyr is shown in Figure 3-4. The class maintains all the necessary metadata, the `current iteration`, the `current step`, `start point`, and `iterations` array. The `iterations` array stores the `iteration` number of each step. Figure 3-3 shows how the `GuideHandler` keeps track of its metadata.

guide decorator

Every Zephyr instance instantiates a `GuideHandler` instance by passing in the ordered key, set, and get methods. The final piece in fully integrating our Guide System with Zephyr is the implementation of the `guide decorator` which wraps each of the relevant Zephyr methods in the `GuideHandler` instance's `guide_step` method. The decorator passes in the method, the method positional arguments, and keyword arguments into the `GuideHandler` instance's `guide_step` method.

The `GuideHandler`'s `guide_step` method is the main method of the class, and

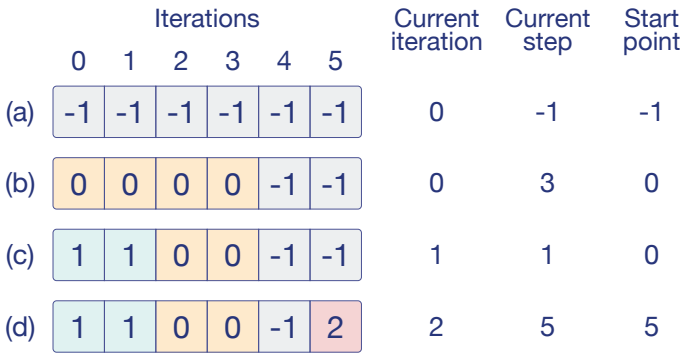


Figure 3-3: `GuideHandler` Metadata Snapshots. This figure shows sequential snapshots of a `GuideHandler`'s `iterations` array, `current iteration`, `current step`, and `start point`, representing a user's potential step flow. The index of the `iterations` array corresponds to the step number. The value at each index corresponds to the `iteration` the step's result is a part of.
(a) This is the initial state. All steps are marked to be at `iteration` -1, current `iteration` is 0, `current step` is -1, and `start point` is -1. If a step's `iteration` is -1, it indicates that the step has never been performed.
(b) From state a to b, steps 0-3 have their `iteration` numbers go from -1 to 0, indicating that they have been performed sequentially in the same `iteration`. When step 0 is performed, whether via `key` or `set` method, `start point` is set to 0.
(c) From state b to c, steps 0 and 1 have gone from `iteration` 0 to 1, indicating that the user has performed a valid backwards step and started a new `iteration`. The user may have performed either both steps 0 and 1, or just step 1. In the latter case, since the `start point` is 0, performing a backwards step at step 1 with a `key` method will update the `iteration` number of step 0.
(d) From state c to d, step 5 has gone from `iteration` -1 to 2, indicating that the user performed step 5 via `set` method, skipping steps 2-4. `Set` methods update the `start point` to the step corresponding to the `set` method and *increment* the `current iteration`. Note that when steps are skipped, their `iteration` numbers do not get updated.

takes in the Zephyr instance, the method the user is attempting to invoke, the method's positional arguments, and the method's keyword arguments. `guide_step` performs checks against the metadata to see whether any warnings need to be logged, then invokes the method if allowed, updates the metadata, and returns the result. Note that the invocation of the Zephyr method within `guide_step` will update any relevant state within the user's Zephyr instance. The implementation of an internal `GuideHandler` class and decorator allows for a clear separation of concern, and prevents any complicated logic unrelated to the Zephyr methods within the methods themselves. Note that the `GuideHandler` class is meant to prevent inconsistencies due to a user's mismanaged steps. Method invocations can still fail due to bad user input, but in that case an error will be raised within the Zephyr method. Since the `GuideHandler`'s state is only updated after the method returns, bad method invocations will prevent any update to the `GuideHandler` state, as desired.

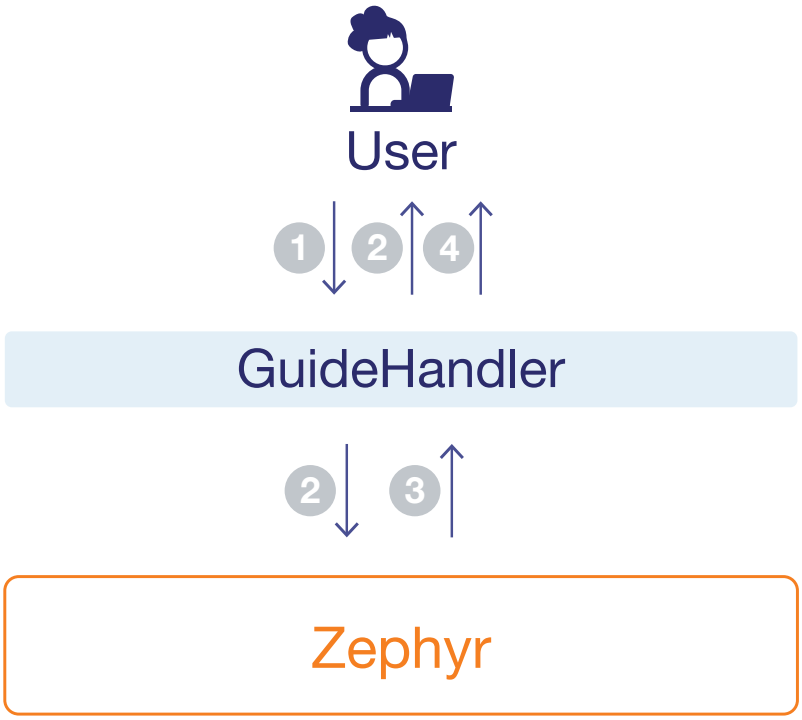


Figure 3-4: `GuideHandler` and Zephyr Overview. (1) User invokes a `Zephyr` method. Note that the user is not aware of the `GuideHandler`, but the diagram shows the user interacting with the `GuideHandler` because the `Zephyr` method is wrapped by `guide_step`. (2) `GuideHandler` checks the method against its metadata, logs a warning to the user if necessary, and invokes the method if the method can be run. (3) If invoked, the `Zephyr` method runs and returns the result to the `guide_step` wrapper. (4) `GuideHandler` updates its metadata and passes the result to the user.

Chapter 4

Discussion

To show that our changes to the Zephyr library reduced fragmentation and improved user experience, we revisit our brake pad case study using the new Zephyr. Using a single Zephyr instance, we will perform the same case study and then go back to previous steps and apply domain knowledge to try to improve model performance. We aim to demonstrate Zephyr's overall improvement to user experience and the usefulness of our Guide System. Assume that all the code listings in this section are run sequentially as they appear, like in a `Jupyter Notebook`, and that therefore every code listing may depend on previous ones.

To begin, we redo all the steps in our background brake pad case study up to fitting the model using the new and improved Zephyr API, as shown in Code Listing 4.1.

```
from zephyr_ml import Zephyr

# create DataFrame dictionary, dfs, from raw data
data_path = 'notebooks/data'
dfs = {
    'turbines': pd.read_csv(os.path.join(data_path, 'turbines.csv'))
        ,
    'alarms': pd.read_csv(os.path.join(data_path, 'alarms.csv')),
    'work_orders': pd.read_csv(os.path.join(data_path, 'work_orders.
        csv')),
```

```
    'stoppages': pd.read_csv(os.path.join(data_path, 'notifications.csv')),
    'notifications': pd.read_csv(os.path.join(data_path, 'notifications.csv')),
    'scada': pd.read_csv(os.path.join(data_path, 'scada.csv'))
}

# initialize Zephyr instance
zephyr = Zephyr()

# create SCADA entityset
zephyr.create_entityset(dfs = dfs, es_type = "scada")

# generate LabelTimes
zephyr.generate_label_times(labeling_fn = "brake_pad_presence")

# generate feature matrix and labels
zephyr.generate_feature_matrixs(target_dataframe_name = "turbines", cutoff_time_in_index = True, agg_primitives = ["count", "sum", "max"], verbose = True)

# generate train test split
zephyr.generate_train_test_split(test_size = 0.3, shuffle = True, stratify = True)

# fit pipeline
zephyr.fit_pipeline(pipeline = "xgb_classifier")
```

Code Listing 4.1: Building a Fitted Model From Scratch. We build a fitted model from raw data using the Zephyr class. Note that here, we disregard outputs from any of the method calls, but it may be common for users to analyze intermediate outputs along the workflow. Refer to the API in the Appendix for further clarification.

4.1 Applying Domain Knowledge

Our current model's performance is decent, with an AUC score of 0.682 as shown in the original case study. To improve this performance, and to demonstrate the process involved, let's incorporate some domain knowledge using Zephyr.

Generate Label Times, Part 2 In the labeling step, there are several parameters we can use to better encapsulate our dataset. Conveniently, we can reuse the same Zephyr instance. Code Listing 4.2 demonstrates recalling the `generate_label_times` method on the same instance with some parameters that encode domain knowledge. (Note that since we are going to a previous step in the same Zephyr instance, the Guide System will log a warning indicating that we are going backwards and creating *stale* steps, as shown in Figure 4-1.)

```
zephyr.generate_label_times(labeling_fn = "brake_pad_presence",
   num_samples = 35, gap = "20d")
```

Code Listing 4.2: Zephyr.generate_label_times with Domain Knowledge. We redo the `generate_label_times` method on the same Zephyr instance, this time with some additional parameters. The `num_samples` is the number of samples that will be created for each instance, so setting it to 35 generates 35 samples per instance. `gap` is the size of the gap between windows in which we sample, so setting it to `20d` means that there are 20 days between our 35 samples of each instance. These additional parameters allow us to check each turbine multiple times across months of data for brake pad issues, and to create a *labeltime* for each instance, better encapsulating our dataset.

Seed Features Moving on to the feature engineering step, let's add some `seed features`, and ignore some data frames that will not be useful for our problem.

`Seed features` are manually defined features that we want to generate, and are more fine-grained than generating `where` features on `interesting values`. They help capture relevant information that comes from domain knowledge, rather than from the raw data alone. For this case study, we will add `seed features` that

```
[GUIDE] STALE WARNING: generate_label_times.
        Going from step 4 to step 1 by performing generate_label_times.
        This is a backwards step via a key method.
        Any results produced by the following steps will be considered stale:
        2. generate_feature_matrix or set_feature_matrix
        3. generate_train_test_split or set_train_test_split
        4. fit_pipeline or set_fitted_pipeline
[GUIDE] DONE: generate_label_times.
        You can perform the next step by calling generate_feature_matrix.
```

Figure 4-1: Example Stale Warning from Guide System. When we perform a valid backwards `key` method, the Guide System tells us the steps that we are making *stale*. After the method is successfully invoked, we are told what the next step should be.

indicate whether or not a brake pad's temperature has exceeded 80 degrees at a certain *label time*. This is a metric known by domain experts to be a potential indicator of impending failure.

To do this, we define the seed features as shown in Code Listing 4.3. We can then pass the seed features, along with some updated parameters, into the `generate_feature_matrix_and_labels` method as shown in Code Listing 4.4.

```
brake_pad_es = zephyr.get_entityset()

seed_features = [
ft.Feature(brake_pad_es["scada"].ww["WROT_Brk1HyTmp1_mean"]) > 80,
ft.Feature(brake_pad_es["scada"].ww["WROT_Brk1HyTmp2_mean"]) > 80,
ft.Feature(brake_pad_es["scada"].ww["WROT_Brk2HyTmp1_mean"]) > 80,
ft.Feature(brake_pad_es["scada"].ww["WROT_Brk2HyTmp2_mean"]) > 80,
]
```

Code Listing 4.3: Example Seed Features. Average brake pad temperature is recorded in the `SCADA` dataset under the columns `WROT_Brk1HyTmp1_mean`, `WROT_Brk1HyTmp2_mean`, `WROT_Brk2HyTmp1_mean`, `WROT_Brk2HyTmp2_mean`, which represent the different average hydraulic oil temperature measurements of different brake pads. We retrieve our *EntitySet* from our Zephyr instance and create a feature for each to indicate whether or not the temperature measurement has exceeded 80 degrees at a certain *label time*.

```
zephyr.generate_feature_matrix(
```

```
    target_dataframe_name = "turbines",
    cutoff_time_in_index = True,
    agg_primitives = ["count", "sum", "max", "percent_true"],
    ignore_dataframes = ["notifications", "alarms", "work_orders"],
    seed_features = seed_features,
    ignore_columns = {"scada": [x for x in brake_pad_es["scada"].
       columns if "Lock" in x]},
    verbose = True
)
```

Code Listing 4.4: Feature Generation with Seed Features. Here we pass in seed features into the `seed_features` parameter of the `zephyr.generate_feature_matrix_and_labels` method of the Zephyr instance we have already used to create a fitted model. We make some additional changes to the parameters of our initial call. We include an additional `percent_true` primitive, which calculates the percentage of True values in boolean columns, to `agg_primitives`. We indicate that the `notifications`, `alarms`, and `work_orders` *DataFrames* are not relevant for feature generation via the `ignore_dataframes` parameter. Finally, we indicate that any columns in the `SCADA` *DataFrame* that contain the string "Lock" will not be relevant for feature generation via the `ignore_columns` parameter and some *DataFrame* manipulation.

At this point, we can continue with our iteration and retrain our model with our new feature matrix and labels. Note that the Guide System outputs the next possible forward `key` method we can perform to get our instance back up to date, which we can follow until we reach the end of our predictive engineering workflow.

```
zephyr.generate_train_test_split(test_size = 0.3, shuffle = True,
   stratify = True)
zephyr.fit_pipeline(pipeline = "xgb_classifier")

zephyr.evaluate()
```

Code Listing 4.5: Zephyr's `evaluate`, revisited. We re-split our data and re-train our model, and then evaluate our new model with Zephyr's `evaluate` method with the

```
[GUIDE] INCONSISTENCY WARNING: evaluate
       Unable to perform evaluate because you are performing a key method at step 5 but the result of the previous step, step 4, is stale.
       If you want to use the stale result or already have the data for step 4, you can use set_fitted_pipeline to set the data.
       Otherwise, you can regenerate the data of the previous step by calling fit_pipeline, and then call evaluate again.
```

Figure 4-2: Example Inconsistency Warning From Guide System

accuracy	0.984993
f1	0.975524
recall	0.980668
precision	0.985866

Figure 4-3: Accuracy, Precision, Recall, and F1 Scores of Improved Model

default parameters. This call will run all of Zephyr's provided evaluation metrics.

4.2 Evaluation

We can now revisit the evaluation step, and retrieve the metrics and plots shown in the original case study with a single call. Note that we cannot call `evaluate` right away, since we also need to re-split our new data and re-train our model on the new data. Figure 4-2 shows the Guide System's warning when we attempt to call `evaluate` right after regenerating our feature matrix. Code Listing 4.5 shows the remaining code necessary to bring our instance up to date with a newly fitted model, along with the `evaluate` call. `evaluate` returns a dictionary that maps the name of each evaluation metric to the output(s) of that metric. We split the output into different figures: Figure **??** shows our new accuracy, precision, recall, and F1 scores, Figure 4-4 shows our new confusion matrix, and Figure 4-5 shows our new AUC-ROC plot.

Comparing our new scores with the original scores for our background case study, we see that the new model is able to achieve near perfect scores for accuracy, precision, recall, and F1 scores – a vast improvement over the original scores.

We also see differences in the confusion matrix and the ROC curve. Because we generated far more samples this time, the confusion matrix now shows much higher numbers overall, but a close look reveals that the new model achieves significantly higher numbers of correct false predictions and correct true predictions.

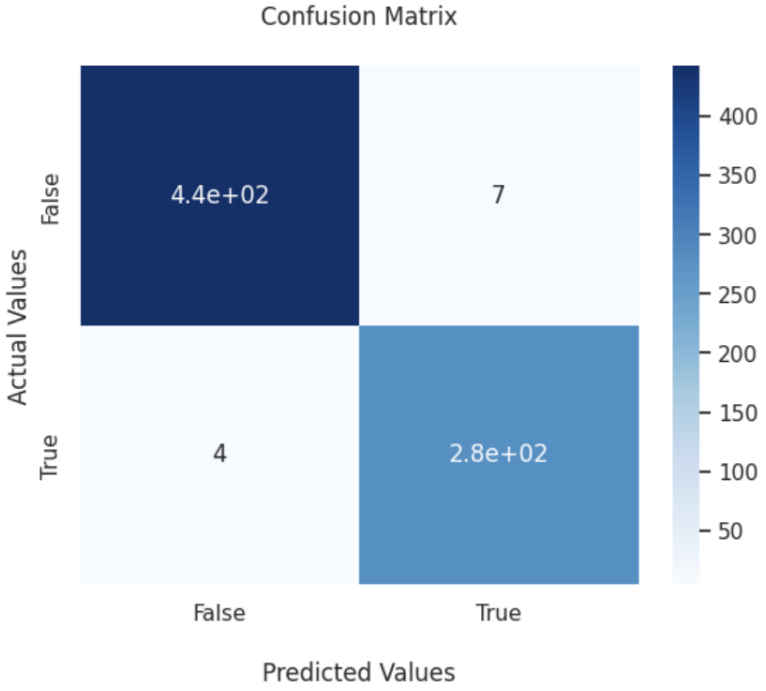


Figure 4-4: Confusion Matrix of Improved Model

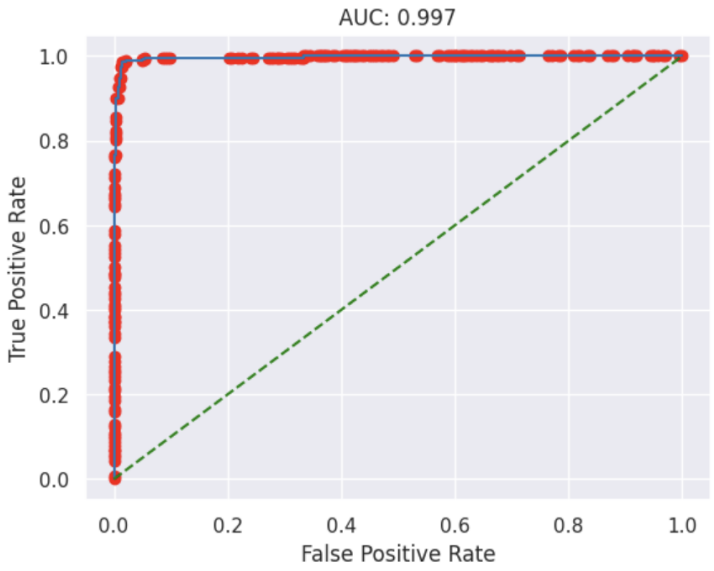


Figure 4-5: AUC-ROC Plot of Improved Model

The ROC curve and AUC score show an almost perfect model. These visuals demonstrate clear improvement over our previous model, emphasizing the importance of domain knowledge for these prediction problems.

Domain knowledge has a large impact on model performance, and the main goal of Zephyr is to enable the application of such knowledge. Our re-done case study demonstrates that our improved Zephyr framework achieves this both more effectively and more efficiently – showing that the new Zephyr is easier for less technical users to work with, that the Guide System is effective, and that the framework uses far less code than previously. We also saw how informative a single `evaluate` call was, despite this simplicity. These improvements to the user experience mean the Zephyr framework can better deliver on its original promise: It lowers the technical barriers that kept many predictive maintenance models in the lab and out of the real world, and it invites the experts who know turbine behavior best to shape, test, and employ those models.

Chapter 5

Conclusion

Zephyr began as solution to the "pipeline jungle" that plagues many predictive maintenance projects in the wind turbine industry and the "black box" solutions that were complex and ultimately unusable by non-machine learning experts. By automating data handling, labeling, feature engineering and prediction engineering, yet still exposing hooks for domain expertise, Zephyr closed much of the gap between one-off research prototypes and production-ready models. Our collaboration with Iberdrola validated Zephyr's approach, but it also revealed fragmentation across functions, hidden states, and a dependence on external tools for key stages of model evaluation.

This thesis addressed those shortcomings through three advances. First, we consolidated the framework behind a single Zephyr API that manages the entire workflow from raw `SCADA`/`PI`/`Vibrations` data to deployable models. Second, we introduced explicit state management with guardrails: our guide system detects stale state, warns users when re-running earlier steps could invalidate later ones, and throws well-scoped exceptions that keep the internal state consistent. Together, these enhancements preserve flexibility for iterative experimentation while preventing the silent divergence during model building that previously led to user confusion. Third, we broadened Zephyr's built-in evaluation suite, adding more metrics and offering a more generalized approach for custom metrics, so that users can better assess model quality without leaving the framework.

5.1 Future Work

Empirically, we have shown that the new Zephyr cuts end-to-end scripting efforts significantly, and provides a lot of clarity throughout the user's workflow. We hope that the new implementation of Zephyr will provide a strong foundation for further development. Our work provides a lot of guidance on user safely reiterating on their model, ensuring that the data is consistent, but there may also be a need in providing guidance on parameter choice and the motivation behind reiterating.

We plan to broaden ZephyrâĂŹs scope by accommodating additional prediction problems and evaluation metrics. Although our new implementation already expands its collection of primitives, further expansion would continue to lighten usersâĂŹ workload. Further, as machine learning research advances and increasingly powerful models emerge, Zephyr should likewise evolve and provide out-of-the-box support for such models.

Appendix A

Zephyr API

```
class Zephyr:
    """
    Zephyr Class.

    The Zephyr Class supports all the steps of the predictive
       engineering workflow for wind farm operations data. It
       manages user state and handles entityset creation, labeling,
       feature engineering, model training and evaluation.
    """

    def GET_ENTITYSET_TYPES(self):
        """
        Get the supported entityset types and their required
           dataframes/columns.

        Returns:
            dict: A dictionary mapping entityset types (PI/SCADA/
               Vibrations) to their descriptions and value.
        """

    def GET_LABELING_FUNCTIONS(self):
        """
        Get the available predefined labeling functions.
```

```
        Returns:
            dict: A dictionary mapping labeling function names to
                their descriptions and implementations.
        """

    def GET_EVALUATION_METRICS(self):
        """
        Get the available evaluation metrics.

        Returns:
            dict: A dictionary mapping metric names to their
                descriptions and MLBlock instances.
        """

    def generate_entityset(self, dfs, es_type, custom_kwargs_mapping
        =None, signal_dataframe_name=None, signal_column=None,
        signal_transformations=None, signal_aggregations=None,
        signal_window_size=None, signal_replace_dataframe=False, **
        sigpro_kwargs):
        """
        Generate an entityset from input dataframes with optional
            signal processing.

        Args:
            dfs (dict): Dictionary mapping entity names to pandas
                DataFrames.
            es_type (str): Type of signal data, either 'SCADA' or '
                PI'.
            custom_kwargs_mapping (dict, optional): Custom keyword
                arguments for entityset creation.
            signal_dataframe_name (str, optional): Name of dataframe
                 containing signal data to process.
            signal_column (str, optional): Name of column containing
                 signal values to process.
            signal_transformations (list[dict], optional): List of
                transformation primitives to apply.
```

```
            signal_aggregations (list[dict], optional): List of
                aggregation primitives to apply.
            signal_window_size (str, optional): Size of window for
                signal binning (e.g. '1h').
            signal_replace_dataframe (bool, optional): Whether to
                replace original signal dataframe.
            **sigpro_kwargs: Additional keyword arguments for signal
                processing.

        Returns:
            featuretools.EntitySet: EntitySet containing the
                processed data and relationships.

        Raises:
            ValueError: If the entityset type is invalid OR if the
                entityset does not follow the data input rules.
        """

    def set_entityset(self, es_type, entityset=None, entityset_path=
        None, custom_kwargs_mapping=None):
        """
        Set the entityset for this Zephyr instance.

        Args:
            es_type (str): The type of entityset (pi/scada/
                vibrations).
            entityset (featuretools.EntitySet, optional): An
                existing entityset to use.
            entityset_path (str, optional): Path to a saved
                entityset to load.
            custom_kwargs_mapping (dict, optional): Custom keyword
                arguments for validation.

        Raises:
            ValueError: If no entityset is provided through any of
                the parameters OR
```

```
            if the entityset type is invalid OR
            if the entityset does not follow the data input
               rules.
    """

def get_entityset(self):
    """
    Get the current entityset.

    Returns:
        featuretools.EntitySet: The current entityset.

    Raises:
        ValueError: If no entityset has been set.
    """

def generate_label_times(self, labeling_fn, num_samples=-1,
    subset=None, column_map={}, verbose=False, thresh=None,
    window_size=None, minimum_data=None, maximum_data=None, gap=
    None, drop_empty=True, **kwargs):
    """
    Generate label times using a labeling function.

    This method applies a labeling function to the entityset to
       generate labels at specific timestamps. The labeling
       function can be either a predefined one (specified by
       name) or a custom callable.

    Args:
        labeling_fn (callable or str): Either a custom labeling
           function or the name of a predefined function (e.g. '
           brake_pad_presence'). Predefined functions like
           brake_pad_presence analyze specific patterns in the
           data (e.g. brake pad mentions in stoppage comments)
           and return a tuple containing:
            1) A label generation function that processes data
```

```
                slices
            2) A denormalized dataframe containing the source
                data
            3) Metadata about the labeling process (e.g. target
                entity, time index)
        num_samples (int, optional): Number of samples to
            generate. -1 for all. Defaults to -1.
        subset (int or float, optional): Number or fraction of
            samples to randomly select.
        column_map (dict, optional): Mapping of column names for
             the labeling function.
        verbose (bool, optional): Whether to display progress.
            Defaults to False.
        thresh (float, optional): Threshold for label
            binarization. If None, tries to use threshold value
            from labeling function metadata, if any.
        window_size (str, optional): Size of the window for
            label generation (e.g. '1h'). If None, tries to use
            window size value from labeling function metadata, if
             any.
        minimum_data (str, optional): Minimum data required
            before cutoff time.
        maximum_data (str, optional): Maximum data required
            after cutoff time.
        gap (str, optional): Minimum gap between consecutive
            labels.
        drop_empty (bool, optional): Whether to drop windows
            with no events. Defaults to True.
        **kwargs: Additional arguments passed to the label
            generation function.

    Returns:
        tuple: (composeml.LabelTimes, dict) The generated label
            times and metadata.
             Label times contain the generated labels at specific
                 timestamps.
```

```
            Metadata contains information about the labeling
                process.

    Raises:
        ValueError: If labeling_fn is a string but not a
            recognized predefined function.
        AssertionError: If entityset has not been generated or
            set or labeling_fn is
             not a string and not callable.
    """

def set_label_times(self, label_times, label_col_name, meta=None
    ):
    """
    Set the label times for this Zephyr instance.

    Args:
        label_times (composeml.LabelTimes): Label times.
        label_col_name (str): Name of the label column.
        meta (dict, optional): Additional metadata about the
            labels.
    """

def get_label_times(self, visualize=False):
    """Get the current label times.

    Args:
        visualize (bool, optional): Whether to display a
            distribution plot. Defaults to False.

    Returns:
        tuple: (composeml.LabelTimes, dict) The label times and
            metadata.
    """

@guide
```

```
    def generate_feature_matrix(self, target_dataframe_name=None,
        instance_ids=None, agg_primitives=None, trans_primitives=None
        , groupby_trans_primitives=None, allowed_paths=None,
        max_depth=2, ignore_dataframes=None, ignore_columns=None,
        primitive_options=None, seed_features=None, drop_contains=
        None, drop_exact=None, where_primitives=None, max_features
        =-1, cutoff_time_in_index=False, save_progress=None,
        features_only=False, training_window=None, approximate=None,
        chunk_size=None, n_jobs=1, dask_kwargs=None, verbose=False,
        return_types=None, progress_callback=None,
        include_cutoff_time=True, add_interesting_values=False,
        max_interesting_values=5, interesting_dataframe_name=None,
        interesting_values=None, signal_dataframe_name=None,
        signal_column=None, signal_transformations=None,
        signal_aggregations=None, signal_window_size=None,
        signal_replace_dataframe=False, **sigpro_kwargs):
        """
        Generate a feature matrix using automated feature
            engineering. Note that this method creates a deepcopy of
            the generated or set entityset within the Zephyr instance
             before performing any signal processing or feature
            generation.

        Args:
            target_dataframe_name (str, optional): Name of target
                entity for feature engineering.
            instance_ids (list, optional): List of specific
                instances to generate features for.
            agg_primitives (list, optional): Aggregation primitives
                to apply.
            trans_primitives (list, optional): Transform primitives
                to apply.
            groupby_trans_primitives (list, optional): Groupby
                transform primitives to apply.
            allowed_paths (list, optional): Allowed entity paths for
                 feature generation.
```

```
            max_depth (int, optional): Maximum allowed depth of
                entity relationships. Defaults to 2.
            ignore_dataframes (list, optional): Dataframes to ignore
                during feature generation.
            ignore_columns (dict, optional): Columns to ignore per
                dataframe.
            primitive_options (dict, optional): Options for specific
                primitives.
            seed_features (list, optional): Seed features to begin
                with.
            drop_contains (list, optional): Drop features containing
                these substrings.
            drop_exact (list, optional): Drop features exactly
                matching these names.
            where_primitives (list, optional): Primitives to use in
                where clauses.
            max_features (int, optional): Maximum number of features
                to return. -1 for all.
            cutoff_time_in_index (bool, optional): Include cutoff
                time in the index.
            save_progress (str, optional): Path to save progress.
            features_only (bool, optional): Return only features
                without calculating values.
            training_window (str, optional): Data window to use for
                training.
            approximate (str, optional): Approximation method to use
                .
            chunk_size (int, optional): Size of chunks for parallel
                processing.
            n_jobs (int, optional): Number of parallel jobs.
                Defaults to 1.
            dask_kwargs (dict, optional): Arguments for dask
                computation.
            verbose (bool, optional): Whether to display progress.
                Defaults to False.
            return_types (list, optional): Types of features to
```

```
            return.
        progress_callback (callable, optional): Callback for
            progress updates.
        include_cutoff_time (bool, optional): Include cutoff
            time features. Defaults to True.
        add_interesting_values (bool, optional): Add interesting
             values. Defaults to False.
        max_interesting_values (int, optional): Maximum
            interesting values per column.
        interesting_dataframe_name (str, optional): Dataframe
            for interesting values.
        interesting_values (dict, optional): Pre-defined
            interesting values.
        signal_dataframe_name (str, optional): Name of dataframe
             containing signal data.
        signal_column (str, optional): Name of column containing
             signal values.
        signal_transformations (list, optional): Signal
            transformations to apply.
        signal_aggregations (list, optional): Signal
            aggregations to apply.
        signal_window_size (str, optional): Window size for
            signal processing.
        signal_replace_dataframe (bool, optional): Replace
            original signal dataframe.
        **sigpro_kwargs: Additional arguments for signal
            processing.

    Returns:
        tuple: (pd.DataFrame, list, featuretools.EntitySet)
            Feature matrix, feature definitions, and the
            processed entityset.
    """


def get_feature_matrix(self):
```

```
        """
        Get the current feature matrix.

        Returns:
            tuple: (pd.DataFrame, str, list) The feature matrix,
                label column name, and feature definitions.
        """


    def set_feature_matrix(self, feature_matrix, labels=None,
        label_col_name="label"):
        """
        Set the feature matrix for this Zephyr instance.

        Args:
            feature_matrix (pd.DataFrame): The feature matrix to use
                .
            labels (array-like, optional): Labels to add to the
                feature matrix.
            label_col_name (str, optional): Name of the label column
                . Defaults to "label".
        """

    def generate_train_test_split(self, test_size=None, train_size=
        None, random_state=None, shuffle=True, stratify=False):
        """
        Generate a train-test split of the feature matrix.

        Args:
            test_size (float or int, optional): Proportion or
                absolute size of test set.
            train_size (float or int, optional): Proportion or
                absolute size of training set.
            random_state (int, optional): Random seed for
                reproducibility.
            shuffle (bool, optional): Whether to shuffle before
```

```
            splitting. Defaults to True.
        stratify (bool or list, optional): Whether to maintain
            label distribution. If True, uses labels for
            stratification. If list, uses those columns. Defaults
             to False.

    Returns:
        tuple: (X_train, X_test, y_train, y_test) The split
            feature matrices and labels.
    """

def set_train_test_split(self, X_train, X_test, y_train, y_test)
    :
    """
    Set the train-test split for this Zephyr instance.

    Args:
        X_train (pd.DataFrame): Training features.
        X_test (pd.DataFrame): Testing features.
        y_train (array-like): Training labels.
        y_test (array-like): Testing labels.
    """

def get_train_test_split(self):
    """
    Get the current train-test split.

    Returns:
        tuple or None: (X_train, X_test, y_train, y_test) if
            split exists, None otherwise.
    """

def set_fitted_pipeline(self, pipeline):
    """
    Set a fitted pipeline for this Zephyr instance.
```

```
        Args:
            pipeline (MLPipeline): The fitted pipeline to use.
        """

    def fit_pipeline(self, pipeline="xgb_classifier",
        pipeline_hyperparameters=None, X=None, y=None, visual=False,
        **kwargs):
        """
        Fit a machine learning pipeline.

        Args:
            pipeline (str or dict or MLPipeline, optional): Pipeline
                to use. Can be:
                - Name of a registered pipeline (default: "
                   xgb_classifier")
                - Path to a JSON pipeline specification
                - Dictionary with pipeline specification
                - MLPipeline instance
            pipeline_hyperparameters (dict, optional):
               Hyperparameters for the pipeline.
            X (pd.DataFrame, optional): Training features. If None,
               uses stored training set.
            y (array-like, optional): Training labels. If None, uses
                stored training labels.
            visual (bool, optional): Whether to return visualization
                data. Defaults to False.
            **kwargs: Additional arguments passed to the pipeline's
               fit method.

        Returns:
            dict or None: If visual=True, returns visualization data
                dictionary.
        """

    def get_fitted_pipeline(self):
        """
```

```
        Get the current fitted pipeline.

        Returns:
            MLPipeline: The current fitted pipeline.
        """

    def predict(self, X=None, visual=False, **kwargs):
        """
        Make predictions using the fitted pipeline.

        Args:
            X (pd.DataFrame, optional): Features to predict on. If
                None, uses test set.
            visual (bool, optional): Whether to return visualization
                 data. Defaults to False.
            **kwargs: Additional arguments passed to the pipeline's
                predict method.

        Returns:
            array-like or tuple: Predictions, and if visual=True,
                also returns visualization data.
        """

    def evaluate(self, X=None, y=None, metrics=None, global_args=
        None, local_args=None, global_mapping=None, local_mapping=
        None):
        """
        Evaluate the fitted pipeline's performance.

        Args:
            X (pd.DataFrame, optional): Features to evaluate on. If
                None, uses test set.
            y (array-like, optional): True labels. If None, uses
                test labels.
            metrics (list, optional): Metrics to compute. If None,
                uses DEFAULT_METRICS.
```

```
            global_args (dict, optional): Arguments passed to all
                metrics.
            local_args (dict, optional): Arguments passed to
                specific metrics.
            global_mapping (dict, optional): Mapping applied to all
                metric inputs.
            local_mapping (dict, optional): Mapping applied to
                specific metric inputs.

        Returns:
            dict: A dictionary mapping metric names to their
                computed values.
        """
```

Bibliography

[1] Sarah Alnegheimish, Najat Alrashed, Faisal Aleissa, Shahad Althobaiti, Dongyu Liu, Mansour Alsaleh, and Kalyan Veeramachaneni. Cardea: An open automated machine learning framework for electronic health records, 2020.

[2] Sarah Alnegheimish, Dongyu Liu, Carles Sala, Laure Berti-Equille, and Kalyan Veeramachaneni. Sintel: A machine learning framework to extract insights from signals. In *Proceedings of the 2022 International Conference on Management of Data*, pages 1855–1865, 2022.

[3] Frances R. Hartwell. *Zephyr: a Data-Centric Framework for Predictive Maintenance of Wind Turbines*. PhD thesis, MIT, 2019.

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

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

[6] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo, and Dan Dennison. Hidden technical debt in machine learning systems. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015.

[7] 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.