SigPro: Enabling Subject Matter Expert Guidance in Feature Engineering

by

Guanpeng Andy Xu

S.B. in Computer Science and Engineering and Mathematics
Massachussetts Institute of Technology (2022)

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

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2024

© 2024 Guanpeng Andy Xu. 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: Guanpeng Andy Xu
Department of Electrical Engineering and Computer Science
January 29, 2024

Certified by: Kalyan Veeramachaneni
Principal Research Scientist, Thesis Supervisor

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

SigPro: Enabling Subject Matter Expert Guidance in Feature Engineering

by

Guanpeng Andy Xu

Submitted to the Department of Electrical Engineering and Computer Science
on January 29, 2024, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, we detail developments to `SigPro`, a feature engineering library in Python guided by Subject Matter Experts (SMEs). `SigPro` includes a suite of data processing building blocks, or *primitives*, as well as an algorithm to combine primitives to form feature engineering *pipelines*. These pipelines are in turn used to construct features for machine learning.

SMEs, through a low-code interface, have several ways to dictate the feature engineering process. First, subject matter experts can construct a feature engineering pipeline for signal data simply by specifying a sequence of data transformations and aggregations (building blocks); `SigPro` then automatically composes a primitive graph and thus a feature engineering pipeline. Second, subject matter experts can also specify parameters and hyperparameters for each building block through `SigPro`'s user-friendly API. These methods encourage SMEs to incorporate their domain knowledge through informative feature transformations and carefully chosen parameter values.

When existing building blocks fall short, `SigPro` facilitates efficient development of new primitives. To this end, we streamline the process for the contribution of new primitives while ensuring their seamless integration into existing pipelines. These improvements ensure that `SigPro` provides an intuitive yet effective solution where subject matter experts can leverage their domain knowledge to generate relevant, explanatory features that can greatly improve the performance of downstream predictive modeling.

Thesis Supervisor: Kalyan Veeramachaneni
Title: Principal Research Scientist

Acknowledgments

My work on this thesis would not be possible without the contributions of others.

Firstly, I would like to thank my supervisor, Kalyan Veeramachaneni, for suggesting and supervising this project and providing invaluable guidance on the design of `SigPro`. I would also like to thank Sarah Alnegheimish for her continued collaboration and support in the development and release of the `SigPro` library, whose existing work on Sintel and `SigPro` has proven critical to my efforts. Without them, none of this work would have been possible.

I am thankful as well to the Data to AI Lab for providing a wonderful environment in which to conduct my research. In addition, I extend my gratitude to Sofia Koukora and Robert Jones of Iberdrola for insightful discussions and expertise as subject-matter experts. Their feedback has been highly beneficial in motivating my work on this project.

Lastly, I would like to thank my friends and family for supporting and encouraging me in all pursuits, research and beyond.

Contents

1 Introduction **13**

1.1 Introduction to Signal Data 14

1.2 Feature Engineering 15

1.2.1 Automation for Feature Engineering 17

1.2.2 The Need for Input from Subject Matter Experts 18

1.3 Goals of SigPro 20

1.4 Thesis Organization 20

2 User-Driven Feature Engineering **21**

2.1 Related Work 21

2.2 The Machine Learning Bazaar 23

2.2.1 Primitive-Pipeline Design 23

2.2.2 MLBlocks 23

2.3 Sintel 24

3 SigPro **27**

3.1 Goals 27

3.1.1 What SigPro Is 27

3.1.2 What Sigpro Isn't 28

3.2 Library Overview 29

3.3 Introduction to Primitives 29

3.3.1 Primitive Taxonomy 30

3.4 Primitive Implementation 33

3.4.1 The `Primitive` Class . . . 33
3.4.2 Subclasses of `Primitive` . . . 35
3.5 Contributing Primitives . . . 36
3.5.1 Dynamic Primitive Class Creation . . . 39
3.6 Pipelines . . . 40
3.6.1 Linear Pipelines . . . 40
3.6.2 Tree Pipelines . . . 41
3.6.3 Layer Pipelines . . . 42
3.7 Pipeline Implementation . . . 44

4 Using SigPro **49**
4.1 Processing Signals . . . 49
4.1.1 Integration into Larger Pipelines . . . 50
4.1.2 Usage Example . . . 51
4.2 Zephyr . . . 52
4.3 Vibrations Dataset Example . . . 53

5 Discussion **55**
5.1 Why SigPro? . . . 55
5.2 Design Rationale . . . 56

6 Conclusion **59**

A Primitive Interface **61**

B Available Primitives **63**

C Pipeline Interface **65**

D Vibrations Dataset Example Code **67**

List of Figures

1-1 Comparing feature engineering, transformation, and selection. 16

3-1 Inheritance relationships of `Primitive` subclasses. 35

3-2 Initializing and tagging primitives with `SigPro`. 36

3-3 Writing the `Mean` aggregation and recording its JSON annotation. . . 38

3-4 Previewing the JSON annotation of the `Mean` primitive. 38

3-5 Dynamically generating the `Mean` primitive. 39

3-6 An example feature 40

3-7 Basic transformation architecture. 41

3-8 Tree transformation architecture. 42

3-9 Example of a layered transformation architecture. 43

3-10 Building a tree pipeline in `SigPro`. 46

4-1 A full signal feature engineering workflow with `SigPro`. 51

4-2 Importing the Vibrations Dataset. 53

4-3 Forming the `SigPro` pipeline with `build_linear_pipeline`. 53

4-4 Processing the signal and plotting the output features. 54

4-5 Distribution of mean (transformed) feature values across each band. . 54

List of Tables

1.1 Feature engineering definitions. 15

2.1 Comparison of feature engineering library functionalities. 22

3.1 Addressing the needs of subject matter experts in `SigPro`. 29

A.1 Interface of the `Primitive` class. 61

B.1 List of available primitive objects in `SigPro`. 64

C.1 `Pipeline` constructors and factory methods. 65

C.2 `Pipeline` interface. 66

Chapter 1

Introduction

The demand for data-driven decision making has increased in recent years, and with it, the need to develop reliable machine learning (ML) models. In particular, feature engineering – the process of identifying, processing, and extracting informative features from raw data – is an important aspect of modeling that has gained increased importance in machine learning and data science recently. After all, it is rare for data to arrive precisely in the ideal input format for a ML model. Achieving success in any ML task requires engineering features that can delve into the important aspects of the data itself while simultaneously accounting for any unique characteristics of downstream learning algorithms [12]. This is especially true when working with raw signal datasets, which often demand time-domain, frequency-domain, and even frequency-time domain transformations to extract useful information from them.

Meanwhile, automated machine learning (AutoML) frameworks have arisen as an approach to traditional machine learning workflows by automating many otherwise labor-intensive tasks in model development [10]. Thus, AutoML approaches reduce the need for challenging, time-consuming, and ad hoc human design. In doing so, they can enable wider access to efficient, scalable, and high-performing models even to users with minimal machine learning experience. When full automation is not desired, Subject Matter Experts (SMEs) can often tailor AutoML approaches to fit their specific domain needs, striking a balance between convenience and customizability.

In this thesis, we focus on feature engineering: specifically, engineering and ex-

tracting features from raw signal data. First, we give an introduction to signal data in Section 1.1 and their unique properties. In Section 1.2, we examine tools for automating feature engineering, as well as the importance of subject matter expert input within this process. We summarize the objectives of our work in Section 1.3 and lay out the rest of the thesis in Section 1.4.

1.1 Introduction to Signal Data

Given the extent of work related to feature engineering with time series [2, 7, 10], it is useful to draw a distinction between signal data and time series data. Time series data, as colloquially referred to in the research community, is typically collected at a regular frequency with a single clean observation at each time step (e.g. stock market data). This data is often aperiodic and information-rich.

On the other hand, signal data is gathered at regular or irregular intervals, usually from sensors. In a typical application, each sensor can collect samples quite rapidly (10^6 samples per second) and assign a series of these readings to a single timestamp. Importantly, signal data is often data-rich but information-poor [3], so multiple signals must often be combined to capture all available information.

Both signal and time series data arrive as a series of time-stamped observations and typically require sequence transformations to generate high-quality features. Nonetheless, the idiosyncrasies of real-world signal data compared to regular time-series mean that additional domain-specific knowledge is essential to extract information from them.

> A real-world example of the kind of signal data we would like to process is the converter failure dataset studied by Hartwell [9]. Among other information, the converter failure dataset contains over twenty million rows of timestamped plant information (PI) signal data. Each row contains numerical information on various wind turbine variables and signals, such as active power, current, and voltage [9]. This PI data can be queried/sampled at regular intervals, as with time series, but ultimately must be transformed with signal processing methods (such as a discrete Fourier Transform) and aggregated before it can be incorporated into a larger converter failure prediction model. The choice of which specific signal processing methods to apply is greatly influenced by

domain knowledge and highlights the difference between signal and time series data in general.

1.2 Feature Engineering

Currently, there are several main approaches employed by practitioners that represent three sub-problems within feature engineering: feature selection, feature extraction, and feature construction [10]. Feature selection works with an existing feature set by removing redundant features and preserving relevant ones, typically employing a search strategy among feature subsets (e.g. forward selection [6]); on the other hand, feature construction is far more dependent on human expertise within the relevant domain[10]. In the case of feature extraction and construction, or feature transformation, a typical approach has been to transform the raw feature set to produce robust, informative, and generalizable features for use in downstream modeling; such transformations can be predefined or manually customized. As early as 2002, Motoda and Liu suggested applying various operators to existing features to produce modified and compound features [14], and many subsequent libraries have extended this principle to contribute their own feature construction paradigms [10].

Term	Definition
Feature engineering	Producing, transforming, and selecting features from the raw data.
Feature selection	Removing irrelevant or redundant features from the data.
Feature transformation	Building new features by carefully transforming the existing features.
Feature extraction	Executing programs to compute features from data
Feature construction	Writing programs that when applied to data can extract features.

Table 1.1: Definitions of important terms related to feature engineering.

We summarize the definition of these terms in Table 1.1. Again, we observe that feature extraction and feature construction are both encompassed by feature transformation, which is in turn an important aspect of feature engineering.

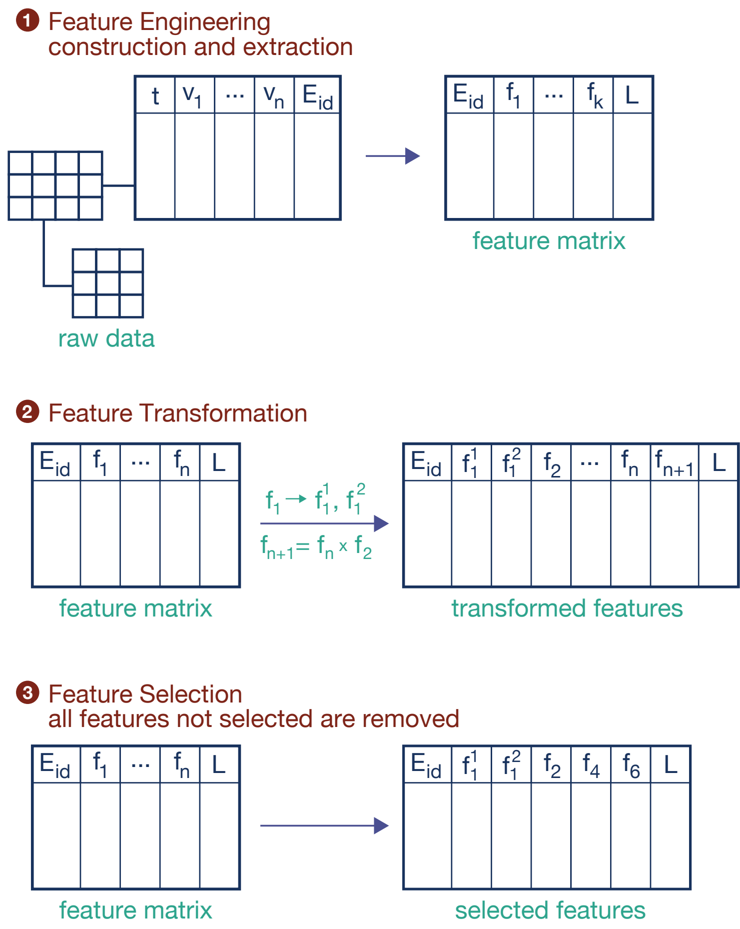


Figure 1-1: A schematic comparison of feature engineering (with feature construction and extraction), feature transformation, and feature selection applied to data.

1.2.1 Automation for Feature Engineering

Several libraries have aimed to automate feature engineering. `Featuretools`, developed in 2015 by Kanter and Veeramacheneni,[1] is a Python library which automatically extracts features from temporal relational data [4, 11]. In their original work, Kanter and Veeramachaneni directly attempt to tackle the need for human-intuition in the feature-engineering stage with the *Deep Feature Synthesis* (DFS) algorithm as part of an end-to end *Data Science Machine*, with a goal to match or exceed human-level performance in data science competitions.

The core data structure of `Featuretools` is the `EntitySet`, which represents a collection of dataframes and their relationships. The DFS algorithm extracts three types of features – entity features, direct features, and relational features – using entity and relational function primitives[2] [11]. To generate each feature, several transformation and aggregation primitives are automatically stacked as dictated by the DFS algorithm, taking into account the relational nature of the data; the user can also manually provide *seed* features to encode domain knowledge into the input of deep feature synthesis. While the DFS algorithm itself and many of the `Featuretools` operations hinge on the relational structure of the intended dataset, the choice to engineer features with a sequence of basic operations is quite relevant to `SigPro` itself.

Other Python libraries of note include `TSFresh`,[3] developed in 2018. `TSFresh` aims to extract features from numerical time series, and is therefore naturally applicable to many signal-processing contexts. `TSFresh` is user-configurable to select varying numbers of features, and as of the latest version, provides several dozen feature calculations within the `tsfresh.feature_extraction.feature_calculators` submodule to produce over 1200 distinct total features [7]. These calculators span a variety of statistical, autoregressive, informational, and even spectral operations. We observe that `TSFresh` generates its features in a parameter-delineated (not modular)

[1] https://www.featuretools.com/

[2] Primitives are reusable software components. We elaborate further on primitives in Chapter 2.

[3] https://github.com/blue-yonder/tsfresh/

fashion, and explicitly selects features from among them via an additional statistical hypothesis test.

Similarly to `TSFresh`, ExploreKit,[4] developed in 2016 by Katz, Shin, and Song, is a feature engineering toolkit that also incorporates feature selection [12]. ExploreKit develops a modular framework in Java that applies a set of unary operations on single features and higher-order operations combining multiple features to generate a candidate feature set. As with `TSFresh`, ExploreKit selects features statistically, but it goes a step further in explicitly comparing performance of models augmented with candidate features to perform its feature ranking [12].

`SigPro`, as we will see later, is designed to *combine* automated feature extraction with subject matter expert input, and none of the aforementioned libraries directly address the use case we hope to tackle in this thesis.

1.2.2 The Need for Input from Subject Matter Experts

A common AutoML approach to low-code feature engineering is to automatically extract a large number of features from the data and to then perform feature selection to produce the final feature set. While such methods can produce acceptable results, they suffer from several problems. First, the amount of dimensionality reduction needed is generally quite drastic due to the sheer quantity of irrelevant and redundant features generated. This results in a significant amount of wasted computation. Secondly, the selected feature set often suffers from low interpretability, limiting the ability of subject matter experts to contribute domain-specific information. Therefore, it is essential to understand the needs of subject matter experts in a feature engineering context.

Most critically, SMEs seek to guide or dictate the feature engineering process by incorporating their domain knowledge. There are several key methods by which SMEs can do so:

(M1) Specify parameter values. At the simplest level, a subject matter expert

[4] https://github.com/giladkatz/ExploreKit

would like to input their knowledge of reasonable parameter ranges by tuning individual parameters (e.g. choosing an energy band for an aggregation).

(M2) Select useful operations. More broadly, subject matter experts are more likely to know which feature transformations are most likely to produce relevant feature outputs, and which transformations do not make intuitive sense based on the nature of the data. For example, a SME familiar with a sensor dataset will know whether or not to apply a frequency-domain transformation such as FFT to the signal values.

(M3) Dictate combinations of operations. Typical features are often the result of applying multiple feature transformations and aggregations in sequence to the raw data (e.g. the mean value of a frequency spectrum). A subject matter expert can leverage their knowledge to focus on only the most promising combinations and pass this information, implicitly or explicitly, to the feature generator.

(M4) Contribute custom operations. Lastly, subject matter experts are well suited to identifying the gaps in existing feature transformations and writing their own operations as needed. Some custom operations, such as the `BandMean` aggregation, are initially written by SMEs but are used widely enough to justify their inclusion in feature engineering libraries.

None of these approaches should require the user to have extensive software engineering experience.

As multiple case studies indicate [9, 13], domain knowledge remains an invaluable input to machine learning systems. Ultimately, an ideal library should enable all avenues of SME contributions within the feature engineering process with a high level, low code interface.

1.3 Goals of SigPro

The goal of `SigPro` is to empower SMEs in signal processing to effectively apply their expertise for feature engineering with an intuitive, flexible interface. `SigPro` enables broad customization of feature engineering pipelines and encourages open-source contributions to maintain its greater relevance and utility to the signal processing community.

1.4 Thesis Organization

In the rest of this thesis, we will present the `SigPro` library and highlight contributions to the primitive and pipeline interfaces. We begin in Chapter 2 with some discussion and related progress in signal and automated feature engineering. In Chapter 3, we examine the `SigPro` framework itself and its structure. We investigate real-world usage and applications of `SigPro` in Chapter 4. Finally, we review our design choices in Chapter 5 and conclude in Chapter 6 by summarizing work on the project thus far and detailing potential updates in the future.

Chapter 2

User-Driven Feature Engineering

In this chapter, we review important prior work that has motivated the design and construction of our library. Our focus is on libraries that promote *user-driven* feature engineering processes.

2.1 Related Work

One of the most widely used general-purpose machine learning libraries in use is `scikit-learn` [1] (also known as `sklearn`), which provides a variety of machine learning algorithms built on the `numpy` and `scipy` libraries [15]. These algorithms span data pre-processing through model fitting and evaluation, including a feature extraction module applicable to image and text datasets. In general, `sklearn` objects are designed to implement a consistent interface depending on their functionality, which allows them to be compatible with both other internal classes as well as external libraries. For example, 'transforms' in `sklearn` implement the 'fit' and 'transform' methods, which allows them to be passed in to create `sklearn.pipeline.Pipeline` instances. Therefore, `sklearn` enables users to chain together a variety of transform objects to comprise a single end-to-end pipeline for the application at hand.

Another option for feature-engineering with a greater focus on time series data is

[1] https://scikit-learn.org/stable/

TSFEL,[2] proposed in 2020 by Barandas et al [2]. TSFEL focuses on fast exploratory data analysis and feature extraction and offers over 60 features across statistical, temporal, and spectral domains, each with its own JSON annotation, that can be extracted from a particular signal. Uniquely, TSFEL also provides an online interface in addition to a conventional Python package. While the overall TSFEL pipeline is relatively fixed in structure, the specific choice of features to be extracted can be customized by the user. When default feature functions are insufficient, TSFEL also enables writing custom features and corresponding annotations.

	featuretools	tsfresh	ExploreKit	sklearn	TSFEL	**SigPro**
User-driven engineering	✗	✗	✗	✓	✓	✓
Signal data support	✓	✓	✗	✗	✓	✓
Library of primitives	✓	✓	✓	✓	✓	✓
Custom primitives	✓	✓	✗	✓	✓	✓
Intuitive primitive design	✓	✓	✓	✗	✓	✓
Custom pipelines	✓	✗	✗	✓	✗	✓
Nonlinear pipelines	✗	✗	✗	✗	✗	✓

Table 2.1: Comparison of feature engineering library functionalities.[3] A ✓ denotes support of the corresponding functionality, while a ✗ indicates that the corresponding functionality is not supported.

In Table 2.1, we summarize several relevant functionalities of the libraries we have examined thus far compared to SigPro. While all examined frameworks provide a library of primitive operations to transform raw data, they differ in their intended usage scenarios, pipeline customizability, and simplicity of use.

[2] https://github.com/fraunhoferportugal/tsfel

[3] *User-driven engineering* indicates whether the feature engineering process is dictated primarily by the user or not. *Signal data support* describes the extent of common signal processing operations (e.g. discrete Fourier transform) usable with each library. *Primitives* denote reusable software components which perform a single operation; some libraries provide a *library of primitives* for the user, and certain libraries also enable the user to write *custom primitives*, ideally with an *intuitive primitive design*. To perform a task, the user combines various primitives in some manner to form a *pipeline*. Some libraries explicitly support the creation of user-specified *custom pipelines*, and a subset allow these pipelines to combine primitives in either a *linear* (explicitly sequential) or *nonlinear* pattern.

2.2 The Machine Learning Bazaar

In addition to serving as a standalone toolkit for signal feature engineering, `SigPro` also represents the latest iteration in a series of developments towards a comprehensive software ecosystem for developing robust end-to-end machine learning applications.

2.2.1 Primitive-Pipeline Design

An important pair of concepts introduced by Smith, Sala, Kanter, and Veeramachaneni (2019) are the *primitive* and the *pipeline*. Observing inefficiencies in the traditional ML development process, Smith et al. envisioned the `MLBazaar` to ease the practical construction of ML systems [18]. For our purposes, a *primitive* is simply a reusable software component that processes data input with some operation and returns an output [1]. Meanwhile, a *pipeline* refers to an end-to-end program comprised by primitives. In the context of general end-to-end machine learning tasks, primitives could handle operations ranging from data cleaning and pre-processing to model fitting and prediction [18]; any specific combination of primitives to perform the task could comprise a pipeline.

The primitive-pipeline design pattern has several advantages. Users can freely combine, reuse, and replace primitives without repeatedly integrating third-party tools or writing extensive glue code [18]. Meanwhile, pipelines provide a concise, flexible approach to defining reusable workflows that do not need to be rewritten when any individual component is revised. The result is a code-efficient framework that improves transparency, lowers error potential, and encourages constructive development practices with thorough unit testing and documentation [1].

2.2.2 MLBlocks

First developed in 2015 by Bryan Collazo and extended in 2018 by William Xue, `MLBlocks` [4] provides a framework for constructing end-to-end tunable machine learning pipelines to tackle a variety of data science problems [8, 19]. As first conceived by

[4] https://mlbazaar.github.io/MLBlocks/

Collazo, `MLBlocks` was a software system to allow data scientists to combine reusable software modules, or *blocks*, into end-to-end high level techniques (e.g. discriminative or generative modeling). After extracting, interpreting, and aggregating the data – the feature engineering users would have four possible 'lines' to follow, each corresponding to a single machine learning technique supported by the library.

Xue reintroduces `MLBlocks` as a lightweight Python library to integrate various other machine learning libraries with an intuitive interface. Machine learning in `MLBlocks` relies on the basic `MLBlock` class, which provides an abstraction for a single data science primitive, and the `MLPipeline` class, which allows users to simply tie together `MLBlocks` to form a machine learning pipeline. The `MLBlock` class seamlessly integrates third-party libraries and user-written functions, while the `MLPipeline` class serves as the main user interface for the libary [19].

Just as in `sklearn`, a successful `MLPipeline` requires only that its consituent primitives implement a similar inferface. The user then interacts with the pipeline by applying either its `fit` method or `predict` method to iteratively apply either the `fit` or `produce` methods, respectively, of its `MLBlock` steps. For example, if each `MLBlock` step applies a single feature transformation of an input data series as its `produce` method, the `predict` method of the resulting `MLPipeline` will extract one or more iteratively transformed *features* as its output. Therefore, the modern `MLBlocks` framework can be used to perform feature engineering as well.

2.3 Sintel

`SigPro` and several other libraries together comprise the Signal Intelligence project, or `Sintel`,[5] an open-source, end-to-end framework for performing time series tasks such as Anomaly Detection (AD) [1]. In the AD case, by following the aforementioned primitive-pipeline approach, `Sintel` decomposes the task into three modules: preprocessing, modeling, and post-processing. As a feature engineering library, `SigPro` naturally integrates with the first phase of the process.

[5]https://github.com/sintel-dev/

As before, splitting the AD task allows for the re-use of primitives across pipelines and integrating new primitives without entirely reworking the pipeline itself. `Sintel` provides default pipeline implementations for several state-of-the-art AD methods, including LSTM and ARIMA approaches. For other tasks, users leverage the customizability of `Sintel` pipelines to substitute or compose bespoke workflows at will.

Chapter 3

SigPro

In this chapter, we will express the purpose of the `SigPro` library and discuss the role of primitives and pipelines.

3.1 Goals

`SigPro` is a feature generation library intended to facilitate SME feature engineering for signal datasets. The goal of `SigPro` is to enable SMEs to leverage their domain knowledge in order to engineer their own feature pipelines from signal data, without the need for extensive understanding of deep feature extraction methods or machine learning development in general.

3.1.1 What SigPro Is

As we observed in Section 1.2.2, SMEs have a variety of avenues through which they can impute their domain knowledge. Therefore, `SigPro` is intended to:

- **Streamline open-source development of feature engineering pipelines.** `SigPro` provides a pre-built library of transformations and aggregations that are composed into pipelines to ensure that first-time usage remains rapid and intuitive even for users with less machine learning engineering experience. In particular, `SigPro` should be easy to develop with for domain experts, and its

code should be easy to understand for potential contributors of new primitives.

- **Enable customization of specific primitives and pipeline structures.** In certain cases, the specific use scenarios of SMEs do not coincide perfectly with the available functionality in `SigPro`. Therefore, `SigPro` makes it easy to express custom signal operations and sequences as primitives and pipelines, respectively. Ideally, `SigPro`'s interface will also be familiar to expert users to facilitate community usage and contribution.

- **Provide flexibility in case of changing requirements.** User requirements are not always static and in many situations the user will need to modify a pre-existing pipeline. Thus, `SigPro` supports modifying and composing primitives and/or pipelines without creating a new object from scratch.

3.1.2 What Sigpro Isn't

In contrast, `SigPro` is not intended to:

- **Perform data cleaning or formatting.** `SigPro` expects signal data to arrive in a certain format, and it is the responsibility of the user to collect and sanitize raw data into this form.

- **Eliminate the need for human involvement.** Our goal is not to provide a black-box catch-all automated feature extractor. While `SigPro` facilitates and automates some aspects of feature engineering development, it does not substitute for human intuition and domain expertise.

In short, the domain expert should be leveraging `SigPro` to dictate the generation of features, not vice versa. That said, SMEs can certainly use `SigPro` to guide the development of their own feature engineering pipelines, potentially through exploratory data analysis. Future iterations of `SigPro` could also provide supplementary information (e.g. statistical feature summaries) that can be taken into account by the SME using the library.

3.2 Library Overview

In `SigPro`, just as with `MLBlocks` [19], the user engineers features by composing individual transformations and aggregation primitives to create a pipeline. These primitives can originate from either `SigPro`'s built-in library or can be written and customized by the user, and are designed to be compatible with `MLBlocks`-style JSON specifications. We will delve further into `SigPro` primitives, their taxonomy, and their interface starting in Section 3.3.

Pipelines themselves can be built using several basic structures: linear pipelines (Section 3.6.1), tree pipelines (Section 3.6.2), and layered pipelines (Section 3.6.3). Each pipeline has its own unique associated structure, resulting in different sets of output features.

	SME Need	Feature	Sections
M1	**Specify parameter values**	`Primitive` class	3.4.1 3.4.2
M2	**Select useful operations**	Primitive Taxonomy `Primitive` class Pipelines	3.3.1 3.4 3.6
M3	**Dictate combinations of operations**	Linear Pipelines Tree Pipelines Layer Pipelines Building Pipelines	3.6.1 3.6.2 3.6.3 3.7
M4	**Contribute custom operations**	Primitive Contribution Dynamic Class Creation	3.5 3.5.1

Table 3.1: Addressing the needs of subject matter experts in `SigPro` to tune parameter values **(M1)**, select useful primitives **(M2)**, specify combinations of primitives **(M3)**, and contribute custom primitives when required **(M4)**.

3.3 Introduction to Primitives

In Section 2.2, we defined the notion of primitives and pipelines. Because they enable highly customizable and flexible workflows, primitives and pipelines are critical to the

design of `SigPro`. Therefore, we devote the rest of the chapter to `SigPro`'s realization of the primitive-pipeline concept and its relevance to our library goals.

Primitives comprise the modular building blocks of `SigPro`. In `SigPro`, primitives consist of a primitive *function*, primitive *object*, and associated JSON *annotation*. Most primitive functions transform an input *signal* time series into an output *feature* time series or scalar value and possibly a few additional outputs. We delve further into specific input and output formats later in this chapter. Similarly to `MLBlocks`, information about individual primitives is stored in corresponding JSON *annotations*, which standardize input and output formats to reduce glue code for the user. Finally, the primitive object cleanly encompasses the primitive function and JSON annotation and represents the primary user interface for the primitive.

3.3.1 Primitive Taxonomy

From a user perspective, `SigPro` offers two basic *types* of primitives for use: transformations and aggregations **(M2)**. Each type of primitives is further broken into several primitive *subtypes*, including amplitude, frequency, and frequency-time primitives. Built-in primitive JSON annotations are individually stored according to their `(primitive_type, primitive_subtype)` attributes into the appropriate subfolders.

Transformations

A *transformation* primitive takes as input a series of signal values, along with some optional hyperparameters, and outputs a new series containing an alternative representation of the original data. `SigPro` transformations can be broken into several categories:

1. **Amplitude** transformations accept as input a 1-dimensional `numpy.ndarray` of amplitude values `amplitude_values` and return a single 1-dimensional `ndarray` of `amplitude_values` as output. An example of an amplitude transformation is the `Identity` primitive, which simply outputs the input amplitude values as-is.

2. **Frequency** transformations accept a 1-dimensional `ndarray` of time series values `amplitude_values` and a sampling frequency `sampling_frequency` as inputs, and performs a frequency based transformation to the input series. These transformations will output two 1-dimensional `ndarrays`: `amplitude_values`, containing frequency amplitudes, and `frequency_values`, containing the corresponding frequencies. An example of a frequency transformation is the `FFT` primitive, which computes and returns the discrete Fourier Transform and associated frequency values.

3. **Frequency-time** transformations take the same inputs as frequency transformations and return a 2-dimensional `ndarray` representation `amplitude_values` of the input signal, along with the corresponding 1-dimensional `frequency_values` and `time_values`. The `STFT` transformation, which computes the short-time Fourier Transform, is one such primitive.

`SigPro` transformation primitives are collectively represented by the `TransformationPrimitive` class, which inherits from `Primitive`. Furthermore, amplitude transformations, frequency transformations, and frequency-time transformations are represented by the subclasses `AmplitudeTransformation`, `FrequencyTranformation`, and `FrequencyTimeTransformation`, respectively.

Aggregations

An *aggregation* primitive takes as input a series of signal values (or other series), along with some optional hyperparameters, and outputs one or several scalar values which summarize the data. `SigPro` aggregations can be broken into several categories as well:

1. **Statistical** aggregations accept a `ndarray` of signal values `amplitude_values` as input and apply a statistical function to return an output. We refer to these primitives as *amplitude* aggregations, as they do not work with accompanying frequency values. Two examples of statistical (amplitude) aggregations are the

`Mean` and `Var` aggregations, which respectively compute the mean and variance of the input data.

2. **Spectral** aggregations accept as input two `ndarray`s, `amplitude_values` and `frequency_values`, for more domain-specific feature generation, and return a scalar output. We refer to such primitives as *frequency* aggregations to contrast them with amplitude aggregations. For instance, the `BandMean` primitive computes the mean of the input amplitude values lying within a user-specified frequency interval, and therefore serves as an example of a spectral (frequency) aggregation.

3. **Frequency-Time** aggregations accept three `ndarray`s as input – `amplitude_values`, `frequency_values`, and `time_values` – and return a scalar output. Currently, `SigPro` does not provide any such frequency-time aggregations by default, though the user can freely implement their own such primitives as their use case demands.

4. **Comparative** aggregations accept as input both `amplitude_values` and `reference_values`, and produce a scalar output representing some comparison between the transformed signal and reference `ndarray`s. An example would be a correlation primitive that computes the Pearson correlation between the signal values and a set of reference values. These primitives can also be expressed as amplitude aggregations with the reference values as context arguments to the primitive, so we do not explicitly group them within a separate subclass of `AggregationPrimitive`.

Similarly to transformations, `SigPro` aggregations are collectively implemented under the `AggregationPrimitive` class. Paralleling the transformation nomenclature, we represent statistical (and comparative), spectral, and frequency-time aggregations by the subclasses `AmplitudeAggregation`, `FrequencyAggregation`, and `FrequencyTimeAggregation`, respectively.

3.4 Primitive Implementation

As noted previously, all transformations and aggregations inherit from the `Primitive` base class, which represents a generic `SigPro` primitive. Built-in transformation and aggregation primitives are implemented under a class-subclass hierarchy based on their subtypes, and users are encouraged to implement their own custom primitives under this system as well.

3.4.1 The `Primitive` Class

The `Primitive` class serves as the foundation for all primitives, built-in and user-defined, in `SigPro`. A `Primitive` can be initialized by passing in the extended primitive name `primitive`, which indicates the location of the primitive function itself, as well as the `primitive_type` and `primitive_subtype`; optionally, the user can specify hyperparameter values in dictionary format through the `init_params` parameter **(M1)**. While the initialization syntax of basic primitive objects is slightly involved, we do not expect most users to initialize or subclass from `Primitive` directly. Rather, users can inherit from the subclass that most directly corresponds to the primitive type and subtype of their desired primitive.

In addition to initialization, users can apply several getters and setters to access and modify important attributes of each `Primitive`. Notably, we enable users to set and access primitive *tags*, which allow users to name primitive objects intelligibly and uniquely (in the case of multiple instances of a particular primitive), and fetch the hyperparameter dictionary. Both of these methods are used internally to interface with `SigPro` pipelines. Finally, we provide `make_primitive_json` and `write_primitive_json` instance methods to enable users to easily create and store JSON annotations compatible with the `MLBlock` API. While JSON annotations have already been provided for built-in primitives, these methods are useful for quickly contributing and using custom primitives without writing a JSON annotation by hand. We save a more careful presentation of `make_primitive_json`, `write_primitive_json`, and the primitive contribution process for Section 3.5. The

full public interface is given in Table A.1.

It is worth noting that unlike the analagous `MLBlock` class in `MLBlocks`, `SigPro Primitives` *include their hyperparameter values* within their representation. This allows us to avoid having to separately pass in (fixed) hyperparameter values into our feature engineering pipelines and yields a simpler, more intuitive interface for the end user **(M1)**.

Primitive Attributes

For the sake of clarity, we explicitly enumerate the attributes of `Primitive` that are relevant to producing the JSON annotations, and by extension, the usage of primitives in pipelines.

- `primitive (str)`: The fully qualified `str` name for the primitive function (e.g. `"sigpro.transformations.frequency_time.stft.stft"`).
- `tag (str)`: The user-given 'tag' to a primitive.
- `primitive_type (str)`: The type of a primitive.
- `primitive_subtype (str)`: The subtype of a primitive.
- `fixed_hyperparameters (dict)`: The dictionary of (fixed) hyperparameter names and their types.
- `context_arguments (list)`: The list of context arguments expected by the primitive.

We note a distinction between a (fixed) hyperparameter and a context argument, which both appear as additional arguments in the function header underlying the primitive. The former is a 'fundamental' characteristic of the transformation or aggregation itself, and therefore should be specified upon initialization. In contrast, a context argument is not fundamental to the primitive and is passed in at run-time by the input data.

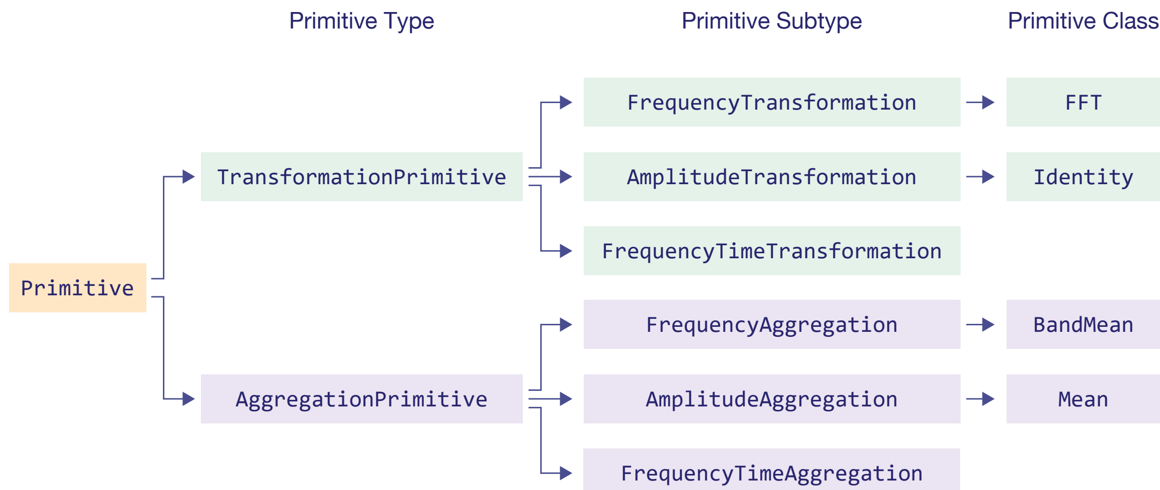


Figure 3-1: Inheritance relationships of `Primitive` subclasses with selected primitives. `Primitive` are broken in two primitive type classes and six primitive subtype classes, which in turn serve as direct superclasses for specific primtiives.

3.4.2 Subclasses of `Primitive`

Recall from Section 3.3.1 that the subclass structure under `Primitive` mirrors the various types and subtypes of primitives that can be written in `SigPro`. Transformation and aggregation primitives are represented by the `TransformationPrimitive` and `AggregationPrimitive` types, and each of these types is further subclassed into various subtype classes, such as `AmplitudeAggregation` and `FrequencyTimeTransformation`. Finally, specific primitives, such as `FFT` and `Mean`, inherit from the proper superclass based on their corresponding type and subtype (Figure 3-1). A full enumeration of available `SigPro` primitive classes is provided in Table B.1.

While the interfaces for all type and subtype classes are similar, the initialization interface of specific primitives can be significantly simplified (Figure 3-2). Indeed, many built-in primitives can be instantiating without specifying any parameters whatsoever. In other cases, where the specific transformation or aggregation to be applied is controlled by at least one hyperparameter input, the user initializes the primitive by passing in hyperparameter values **(M1)** as keyword arguments (e.g. `BandMean(10,30)`). This creation syntax mirrors other machine learning libraries such as `Keras` and increases the readability of the pipeline code as a whole.

```python
from sigpro.basic_primitives import (
    Identity, FFT, FFTReal, Mean, Kurtosis)

identity_tfm = Identity().set_tag('id')
fft_tfm, fft_real_tfm = FFT(), FFTReal()
mean_agg, kurtosis_agg = Mean(), Kurtosis(bias=False)
```

Figure 3-2: Initializing and tagging primitives with `SigPro`. The `identity_tfm` primitive is given the alternative tag `'id'`, while the `kurtosis_agg` primitive is initialized with its `bias` parameter set to `False`.

3.5 Contributing Primitives

An important aspect of `SigPro` is the ability to efficiently define custom primitives and seamlessly integrate them into pipelines **(M4)**. To contribute and use a custom primitive, say, `UserPrimitive`, the user undergoes the following steps.

1. The user writes the primitive function `userprimitive(amplitude_values, **kwargs)` itself. This function can be a wrapper around an existing (third-party) library function, or it can be a custom transformation or aggregation.

2. The user identifies the proper primitive type and subtype that most commonly match the inputs and outputs of the written function, and stores the function within the appropriate `SigPro` folder according to the aforementioned taxonomy.

3. The user then creates a new subclass that inherits from the appropriate `Primitive` subclass and populates the `__init__` method, including any necessary inputs to `init_params` (e.g. hyperparameters) as additional arguments **(M1)**. Alternatively, the user can call the `make_primitive_class` method to dynamically generate a primitive class that can be used in pipelines.

4. For aggregation primitives, the user should call `Primitive.set_primitive_outputs (self, primitive_outputs)` to name the output of the primitive accordingly (e.g. `mean_value` for a `Mean` primitive). This step is not necessary if the user has already customized the output names with `make_primitive_class`.

5. Finally, the user instantiates their primitive with the `UserPrimitive(**kwargs)` syntax, where `kwargs` contains any hyperparameter values passed into the primitive as keyword arguments. If the user has not done so already, they should compute and record the primitive JSON annotation by calling `write_primitive_json`.

In Figures 3-3 and 3-4, we detail the contribution and initiation process as it would apply to writing a typical amplitude aggregation.

Recall that primitives with hyperparameters are instantiated by passing in the hyperparameters as (keyword) arguments themselves. One alternative design for the instantiation interface would be to allow or require the user to directly pass in hyperparameter names in an `init_params` dictionary. While we allow the user to *export* the hyperparameter values in dictionary format, in case the user would like to store or use such hyperparameters as part of a larger pipeline, we choose not to encourage such initialization for typical primitives in `SigPro` with few hyperparameters for readability reasons. If the number of hyperparameters is high and direct keyword initiation is less feasible, we nevertheless encourage the user to accept inputs with the `**kwargs` format and unpack any input dictionaries with `**init_params`.

Additionally, in calling our superclass constructor (itself subclassed from `Primitive`) within `Main.__init__`, we did not pass in the primitive outputs directly. Rather, we explicitly set the primitive outputs with a separate setter method. If we had needed to specify hyperparameters, we would have called the `set_fixed_hyperparameters` method on a subsequent line. Why not attempt to pass in these outputs and hyperparameters to the `AmplitudeAggregation` constructor itself, possibly as optional arguments? Unfortunately, what small improvements this design achieves in brevity of code are outweighed by setbacks in the ease of using the constructor in the first place. In short, we adopt a *use-as-needed* paradigm. Very few primitives in the `SigPro` library would use many or all of the optional arguments thus provided, at the cost of requiring users to familiarize themselves with a litany of additional initialization parameters in `__init__`. Explicitly denominating most of these arguments as *steps* in primitive initialization rather than additional *inputs* to the initialization itself relieves this burden on our end users and enables them to add complexity to

```python
from sigpro import primitive
# Steps 1-2
def mean(amplitude_values): # in sigpro.aggregations.amplitude.statistical.py
    return np.mean(amplitude_values) # i/o matches AmplitudeAggregation
# Step 3
class Mean(primitive.AmplitudeAggregation):
    def __init__(self):
        super().__init__("sigpro.aggregations.amplitude.statistical.mean")
        # Step 4
        self.set_primitive_outputs([{"name": "mean_value", "type": "float" }])
# Step 5
myprimitive = Mean()
myprimitive.write_primitive_json()
```

Figure 3-3: Writing the `Mean` aggregation and recording its JSON annotation.

```python
myprimitive = Mean()
print(myprimititve.make_primitive_json())
```

```
# Output:
{   "name": "sigpro.aggregations.amplitude.statistical.mean",
    "primitive": "sigpro.aggregations.amplitude.statistical.mean",
    "classifiers": {
        "type": "aggregation",
        "subtype": "amplitude"
    },
    "produce": {
        "args": [
            {
                "name": "amplitude_values",
                "type": "numpy.ndarray"
            }
        ],
        "output": [
            {
                "name": "mean_value",
                "type": "float"
            }
        ]
}}
```

Figure 3-4: Previewing the JSON annotation of the `Mean` primitive.

```python
from sigpro.basic_primitives import Mean
from sigpro.contributing_primitive import get_primitive_class

def mean(amplitude_values): # in sigpro.aggregations.amplitude.statistical.py
    return np.mean(amplitude_values)

mean_primitive = "sigpro.aggregations.amplitude.statistical.mean"
mean_outputs = [{'name': 'mean_value', 'type': 'float'}]
MeanDynamic = get_primitive_class(mean_primitive,
                                  'aggregation',
                                  'amplitude',
                                  primitive_outputs=mean_outputs)

mean_dynamic, mean_subclass = MeanDynamic(), Mean()
print(mean_dynamic.make_primitive_json() == mean_subclass.make_primitive_json())

# Output
# True
```

Figure 3-5: Dynamically generating the `Mean` primitive with `get_primitive_class`. While we still need to write the function body, we no longer need to write a class definition. The resulting dynamic primitive class is functionally identical to the explicit subclass implementation.

their primitives only as required.

3.5.1 Dynamic Primitive Class Creation

While we intentionally avoid methods with many input arguments, we recognize that some users are more comfortable with imperative programming than object-oriented programming. Thus, we provide the `get_primitive_class` method to return a dynamically generated primitive class with the same functionality as a correspondingly written subclass implementation; the `make_primitive_class` method is similar but also records the JSON annotation **(M4)**. Figure 3-5 shows an example of dynamic class creation with this method.

3.6 Pipelines

While primitives can be quite useful on their own, the true power of `SigPro` arises in the development of a feature engineering pipeline. Here, SMEs are able to leverage their domain knowledge to construct bespoke feature generation pipelines specific to their individual use cases **(M3)**.

In `SigPro`, *pipelines* denote a set of primitive operations that are combined and/or composed in some form to transform an input time series into one or more output feature values. Because transformation primitives generally map signal time series to signal time series, while aggregations generally map time series to scalar values, output features of effective `SigPro` pipelines will generally result from a series of transformations followed by an output of an aggregation primitive.

Figure 3-6: An example feature produced by applying the `FFT` and `FrequencyBand` transformations, followed by the `Mean` aggregation, to the input array `values`.

For the rest of this section, we will discuss several pipeline *structures* **(M3)**. In order of generality, these structures are *linear* pipelines, *tree* pipelines, and *layer* pipelines. The diversity of available structures mean that the same list of primitives can be combined to extract many different feature outputs.

3.6.1 Linear Pipelines

The most basic form of a `SigPro` pipeline is the *linear* pipeline. A linear pipeline can be specified by an ordered list of transformation primitives $T_1, T_2, \ldots, T_n$ and a (potentially unordered) list of aggregation primitives $A_1, A_2, \ldots, A_m$ **(M2)**. To extract output features, the pipeline applies transformations $T_1, T_2, \ldots, T_n$ in order to obtain a transformed time series, and then applies aggregations $A_1, A_2, \ldots, A_m$ to obtain several aggregated features from this time series; since the aggregations are all applied to the same transformed time series, the order in which we apply the aggregations is immaterial. The result of this process is a set of m output fea-

tures $T_1.T_2.\cdots T_n.A_1, T_1.T_2.\cdots T_n.A_2, \ldots, T_1.T_2.\cdots T_n.A_m$. We summarize this feature transformation architecture in Figure 3-7.

Figure 3-7: Basic (linear) transformation architecture. Each box represents a single (intermediate) output.

3.6.2 Tree Pipelines

Linear SME-curated transformation series are important to `SigPro`, and linear pipelines are able to generate a wide variety of features. However, there are certainly situations in which subject matter experts would like to generate multiple features at once, using multiple transformation sequences **(M3)**. Thus, we also enable more variegated feature generation via *tree* pipelines.

Tree pipelines are characterized by several layers ordered $L_1, L_2, \cdots, L_i, \cdots, L_k$, each containing a list of transformation primitives, and an aggregation layer containing primitives $A = A_1, A_2, \cdots, A_m$ **(M2)**. Suppose that for each $1 \leq i \leq k$ that layer i consists of transformation primitives $T_1^i, T_2^i, \cdots, T_{n_i}^i$. Then for each valid tuple of indices $(t_1, t_2, t_3, \ldots, t_k, a)$ with $1 \leq t_i \leq n_i, 1 \leq a \leq m$, the tree pipeline outputs a feature corresponding to $T_1^{t_1}.T_2^{t_2}.\cdots T_k^{t_k}.A_a$. Ultimately, the tree pipeline generates $m \prod_{i=1}^{k} n_i$ distinct features from the Cartesian product of all primitive layers (Figure 3-8).

Note that when each transformation layer L_i has a single $(n_i = 1)$ transformation, the resulting tree pipeline is equivalent to a linear pipeline. Thus, the tree pipeline is a strict generalization of linear pipelines.

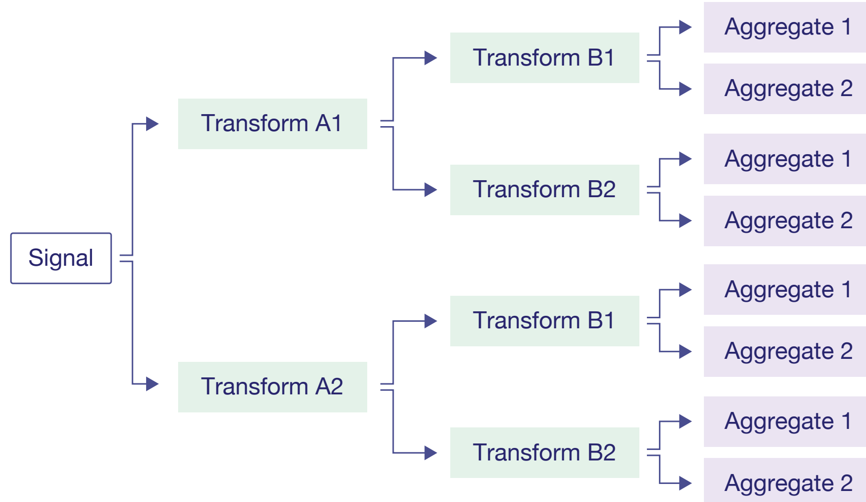


Figure 3-8: Tree transformation architecture. The two transformation layers contain transformation primitives **A1, A2** and **B1, B2**, respectively, while the aggregation layer contains aggregation primitives **Ag1, Ag2**. Each box represents a single (intermediate) output.

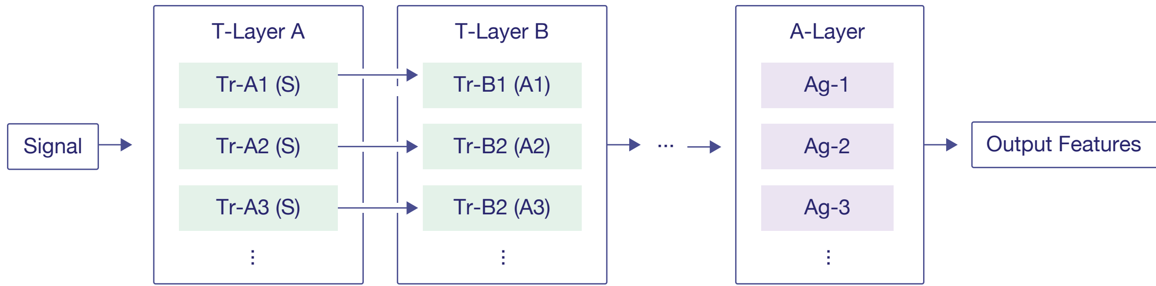
3.6.3 Layer Pipelines

Tree pipelines provide a significantly higher degree of generality for extracting features than their linear counterparts, but such pipelines nevertheless have their own limitations. For instance, producing the full Cartesian product of features in several layers of a tree pipeline may result in more output features than is manageable. Thus, more refined *layer* generation of features is desirable.

Again, suppose we are given several transformation layers $L_1, L_2, \cdots, L_i, \cdots, L_k$ and an aggregation layer containing primitives $A = A_1, A_2, \cdots, A_m$ **(M2)**. However, within each transformation layer i and each transformation T_i^j within layer i, we explicitly specify the set of intermediate outputs from the previous layer that T_i^j is applied to, and similarly for all aggregations in the final layer **(M3)**. Clearly, this generation process results in a subset of features generated by the previous tree pipeline, but does so in a more deliberate fashion (Figure 3-9).

Graph Interpretation

If we interpret the feature generation process as implying a directed graph between nodes corresponding to intermediate outputs, with edges from one output to another




```
1 from sigpro.pipeline import LayerPipeline
2 primitives = [A1, A2, A3, B1, B2, B3, Ag1, Ag2, Ag3]
3 combinations = [(A1, B1, Ag1), (A2, B2, Ag2), (A3, B2, Ag3)]
4 pipeline = LayerPipeline(primitives, combinations)
```

Figure 3-9: Example of a layered transformation architecture with corresponding code. Some generated features could include **A1.B1.Ag1**, **A2.B2.Ag2**, and **A3.B2.Ag3**. Each box represents a single (intermediate) output.

corresponding to the former vertex (output) being used as an input to some primitive to produce the latter vertex (output), the difference between tree and layer pipelines becomes even more clear. With all primitives in L_i and A being held equal, at each layer l, the layer pipeline will produce only a user-specified subset of intermediate outputs $\{T_1^{t_1}.T_2^{t_2}.\cdots T_l^{t_l}\}_{t_1,t_2,\ldots,t_l}$, whereas the tree pipeline will always generate all possible combinations at any layer. Of course, the linear pipeline will only generate a single intermediate output at any layer before the last.

In this discussion thus far, we have not considered the relevance of comparative primitives. Comparative primitives, when applied to a (possibly transformed) time series and a set of reference values, can be conceptualized as simply amplitude aggregations with the reference values passed as an additional context argument. However, there are situations where we would like the reference values to themselves be generated from a series of primitive transformations. Here, we allow the intermediate output nodes corresponding to the comparative primitive to have *multiple* parent nodes, and the rooted-tree structure generalizes to a directed acyclic graph (DAG).

Generality

How general are layer pipelines? Consider any set S of possible output features generated by some number of transformations followed by some aggregation and express it as $S = \{T_{i_1}.T_{i_2}.\cdots T_{i_k}.A_j\}_{i_1,i_2,\ldots,i_k,j}$, where the number k of transformations is held fixed; if the set of output features vary in the number of transformations applied, we can simply pad them with identity transformations until all features have the same length k. Then, at each layer l, we can define layer L_l to be the set of all transformations T_{i_l} that appear as the l-th transformation in some output feature, and define A to be the set of aggregations that appear in the feature set. We can see that the set S of output features can be generated by a suitably chosen set of intermediate outputs at each layer corresponding to the set of all unique prefixes of any length. Hence, we conclude that any set S of features of the desired form can be represented by a general layer pipeline **(M3)**.

3.7 Pipeline Implementation

Just as `SigPro` primitives are represented by specific subclasses of the `Primitive` base class, `SigPro` pipelines are represented by the `Pipeline` abstract base class; however, default `Pipeline` objects cannot be instantiated, and instead `Pipeline` provides for a common interface to process signals (Table C.2).

Interface

By default, `Pipeline` objects possess the following attributes:

- `values_column_name (str)`: The name of the column in the dataframe corresponding to input signal values. Defaults to `"values"`, and is specified in the `process_signal` method.

- `pipeline (mlblocks.MLPipeline)`: The `MLPipeline` to be run on the input data. This should be built by the constructor.

Besides these fields, `Pipeline` objects typically store a list of `transformations` and `aggregations` used. For `MLBlocks` interface purposes, we also record whether the input data is a `DataFrame`.

In addition, the `Pipeline` abstract class implements a variety of getters and setters and specifies the `get_output_features` and `process_signal` methods in its interface. The `get_output_features` method returns a list of features extracted by the pipeline, whereas the `process_signal` method wraps around the constructed `MLPipeline` field stored in `self.pipeline` to perform feature extraction from the input data and output a transformed dataframe. We elaborate on this process further in Section 4.1. Of these two methods, only the `process_signal` method is directly implemented in `Pipeline` itself.

Finally, we provide several factory methods to generate and modify pipelines:

- `build_linear_pipeline(transformations, aggregations)`: Builds a linear pipeline with the list of transformations `transformations` and list of aggregations `aggregations`.

- `build_tree_pipeline(transformation_layers, aggregation_layer)`: Builds a tree pipeline with the list of transformation layers, each a list of primitives, and the aggregation layer, also a list of primitives (Figure 3-10).

- `build_layer_pipeline(primitives, primitive_combinations`: Builds a layer pipeline to generate all of the features in `primitive_combinations` from the primitives given in `primitives`. Each feature in `primitive_combinations` is specified as a tuple of the form $(T_1, T_2, \cdots, T_k, A)$, where $T_1, T_2, \cdots, T_k, A$ are included in `primitives`.

- `merge_pipelines(pipelines)`: Builds a layer pipeline that generates all features produced by the input `pipelines`.

The `merge_pipelines` method gives an easy way for the user to *compose* several existing pipelines (e.g. linear or tree pipelines) into a single, comprehensive feature extractor without directly specifying a list of all desired output features.

```python
from sigpro.basic_primitives import (
    Identity, FFT, FFTReal, Mean, Kurtosis)
from sigpro.pipeline import build_tree_pipeline

identity_tfm = Identity().set_tag('id')
fft_tfm, fft_real_tfm = FFT(), FFTReal()
mean_agg, kurtosis_agg = Mean(), Kurtosis(bias=False)

tfmlayer1 = [identity_tfm]
tfmlayer2 = [fft_tfm, fft_real_tfm]
agglayer = [mean_agg, kurtosis_agg]
tree_pipeline = build_tree_pipeline([tfmlayer1, tfmlayer2], agglayer)
```

Figure 3-10: Building a tree pipeline in `SigPro` from built-in primitives.

Subclasses of `Pipeline`

In `SigPro`, both the `LinearPipeline` and `LayerPipeline` classes implement the `Pipeline` interface.[1] As their names suggest, the `LinearPipeline` class represents linear pipeline structures and the `LayerPipeline` class implements tree and layer pipeline structures. While the representation of linear pipelines is rather straightforward, we opt to internally represent layer pipelines by their output features, rather than their graph structure. The reason for this has already been hinted previously: the generality of our layer pipeline architecture implies that both graph and output-feature representations have the same amount of complexity. In particular, to specify the connections producing the output features ending at the final aggregation layer of a general layer pipeline, the amount of information needed is equivalent to specifying the list of output features to begin with. Moreover, it is computationally quite feasible to switch between these representations using a breadth first search. Finally, this approach greatly simplifies the process of merging multiple pipelines and their produced features into a single `LayerPipeline`.

[1] `SigPro` does not implement a separate `TreePipeline` class as its representation would not be considerably distinct from that of a `LayerPipeline`. `SigPro` does, however, provide three unique *factory* methods with distinct user interfaces for instantiating linear, tree, and layer pipelines.

Building the `MLPipeline`

We previously noted that `Pipeline` objects wrap the `mlblocks.MLPipeline` class in the `self.pipeline` field. However, the pipeline description interface (PDI) for `MLPipeline` objects is typically *sequential* in nature [19, 18]. Thus, some care is needed in creating and using `MLPipeline`s to process signals.

We approach this task in several conceptual stages; the entire `MLPipeline` is built and stored upon initialization of the pipeline.

1. **Ordering the primitives.** We will use the `MLPipeline` to repeatedly transform columns of our input dataframe using primitives that we pass to it. In particular, this means that the order in which `self.pipeline` processes primitives is highly restricted by the dependency relationships in the pipeline structure. For linear pipelines, it suffices to pass in the ordered list of transformations followed by the ordered list of aggregations, since all of the aggregations act upon the result of applying the transformations in sequence. For layer (and tree) pipelines, our execution order can be constructed via a breadth-first traversal of the associated graph. This process is necessarily more involved, and typically involves the repeated execution of a single primitive. Nevertheless, because intermediate outputs are expressed as specifically-named transient columns in the data `DataFrame`, this process allows us to reuse shared computation and produce a more compact pipeline as a whole.

2. **Naming intermediate feature inputs and outputs.** Here, we specify the `input_names` and `output_names` as required by the `MLPipeline` constructor. We name all intermediate outputs in the $\{T_1\}.\{T_2\}\ldots\{T_k\}.k.\{\text{output_name}\}$ format, which are in turn re-used as input names in the following layers. Thus, each primitive expects as input the output of the previous primitive.

   In certain cases, some inputs are context arguments of the primitive, rather than dynamically generated outputs of previous primitives. Such arguments are assigned the input name $\{T\}.\{\text{output_name}\}$ and are expected to be passed within the input Dataframe upon runtime.

3. **Assigning hyperparameter values.** We must explicitly pass in the hyperparameters of each constituent primitive to `init_params`. Fortunately, the `Primitive.get_hyperparam_dict()` method can easily achieve this.

4. **Naming final feature outputs.** Ultimately, we would like our extracted feature names to be intelligible to the user, and reflect the transformation and aggregation primitives that produced them. Therefore, we assign the string name $\{T_1.\text{tag}\}.\{T_2.\text{tag}\}.\ldots.\{T_k.\text{tag}\}.\{A.\text{tag}\}$ to each primitive $T_1.T_2.\cdots.T_k.A$. These values are passed into `output_names` in the `MLPipeline` constructor.

We reiterate that this procedure is completely abstracted away from the user, who needs only to input the desired primitives and primitive combinations to the `build_linear_pipeline` method. Thus, `SigPro` can represent a variety of feature engineering pipelines while maintaining a low-code interface.

After `self.pipeline` is built, we have completed the initialization of the `Pipeline` and are ready to process signal data.

Chapter 4

Using SigPro

Now that we are able to form primitives and construct feature engineering pipelines, we are ready to discuss the true purpose of `SigPro`: to engineer features from real-world signal data.

4.1 Processing Signals

Following our discussion in Section 3.1.2, `SigPro` expects data in the form of a 2-dimensional `pandas.DataFrame` with a specific column, specified by the user, containing the raw signal `amplitude_values`. Each row in the input `DataFrame` should represent a single timestamped observation and contain a single *array* of signal observations in the appropriate values column. Context arguments should be specified as columns with names of the form `{primitive_tag}.{arg_name}` (e.g. `fft.sampling_frequency`).

The `Pipeline.process_signals` method has the following arguments:

- `data (pandas.DataFrame)`: A Dataframe with a signal values column.
- `window (str)`: If specified, the length of the window to resample the signal values so that each entry is itself a time series (e.g. `"1h"`).
- `values_column_name (str)`: The name of the column in the Dataframe corresponding to input signal values.

- `time_index (str)`: The name of the time index column (e.g. `"timestamp"`).
- `groupby_index (Union(str, list[str])`: If specified, the name(s) of the column(s) to group together and take the window over.
- `keep_columns (Union[bool, list]`: If `bool`, whether or not to keep non-feature columns in the output Dataframe. If `list`, the list of the names of columns to be kept in the output Dataframe.
- `input_is_dataframe (bool)`: Whether the input data is a Dataframe. This field is relevant for `MLBlocks` integration but not for signal processing directly.

For a Dataframe with values column named `'values'` and time index `'timestamp'`, only the `data` parameter is required. In turn, `process_signals` returns two outputs:

- `(pandas.DataFrame)` A Dataframe with additional feature columns resulting from applying the pipeline to the data, in additional to the index and whichever non-feature columns are indicated by `keep_columns`.
- `(list)` A list containing the feature names generated by the pipeline.

4.1.1 Integration into Larger Pipelines

Recall from Section 2.3 that `SigPro` pipelines function as feature-extracting preprocessors within an end-to-end machine learning workflow. Because of this, we may be interested in incorporating a full `Pipeline` as a single stage within a larger `SigPro` pipeline, or even an `MLPipeline`. In the former case, the nested `SigPro` pipeline functions as would an aggregation primitive, and the functionality of a `LayerPipeline` is sufficient to accommodate these structures. For the latter situation, `SigPro` currently provides a JSON annotation and `MLBlocks` integration support for the `LinearPipeline` class by way of the `get_input_args` and `get_output_args` methods; support for layer pipelines is planned in the future. Alternatively, `SigPro` users can re-use output features as *seed features* in other feature engineering libraries.

4.1.2 Usage Example

To summarize our presentation of `SigPro` pipelines, we give a full signal-processing example building upon our code in Figures 3-2 and 3-10 with `Primitives` and `Pipelines`. In Figure 4-1, we create and apply a tree pipeline containing three transformations and two aggregations to extract four output features from a demonstration dataset.

```
1 # Import packages
2 from sigpro.basic_primitives import (
3     Identity, FFT, FFTReal, Mean, Kurtosis)
4 from sigpro.demo import _load_demo as get_demo
5 from sigpro.pipeline import build_tree_pipeline
6
7 # Define primitive objects
8 identity_tfm = Identity().set_tag('id')
9 fft_tfm, fft_real_tfm = FFT(), FFTReal()
10 mean_agg, kurtosis_agg = Mean(), Kurtosis(bias=False)
11
12 # Instantiate tree pipeline
13 tfmlayer1 = [identity_tfm]
14 tfmlayer2 = [fft_tfm, fft_real_tfm]
15 agglayer = [mean_agg, kurtosis_agg]
16 tree_pipeline = build_tree_pipeline([tfmlayer1, tfmlayer2], agglayer)
17
18 # Import and reformat sample input data
19 data = get_demo()  # signal values contained in column `values`
20 data['fft.sampling_frequency'] = data['sampling_frequency']
21 data['fft_real.sampling_frequency'] = data['sampling_frequency']
22
23 # Process signal column of sample dataset `data`
24 processed_data, feature_columns = tree_pipeline.process_signal(data)
25 print(feature_columns)
26
27 # Output:
28 # ['id.fft.mean.mean_value', 'id.fft.kurtosis.kurtosis_value',
29 # 'id.fft_real.mean.mean_value', 'id.fft_real.kurtosis.kurtosis_value']
```

Figure 4-1: A full signal feature engineering workflow with `SigPro`.

4.2 Zephyr

In this section, we discuss an application of the SigPro library for signal feature engineering in a real-world wind energy context. Zephyr[1], first released in 2022 by Frances Hartwell, is a 'data-centric framework for predictive maintenance of wind turbines' [9]. To achieve this goal, Zephyr must be flexible to constantly changing infrastructure and requirements while simultaneously integrating domain knowledge at every step of the process. Existing task-specific ML tools, though powerful, fall short of an easy-to-use solution for SMEs looking to create and end-to-end domain-specific model. Instead, Zephyr provides SMEs with a low-code framework spanning the entire scope of data to model development.

As with any ML workflow, Zephyr must extract relevant features from the processed input data. In particular, signal processing techniques are usually required to transform the turbine data, which often arrives in time-series form. Given the similarity in design and usage goals between Zephyr and SigPro, the latter library is an ideal solution to this task. Once the user produces the Dataframe containing signal values, they can specify a set of transformation and aggregation primitives to Zephyr, which will then process the signal column by building the corresponding SigPro pipeline object and calling process_signal. During this phase, the user is free to explore, use, or even contribute their own primitive combinations with SigPro's primitive and pipeline API. [2]

Once the time series data is transformed and aggregated into several feature columns, the user can proceed by applying other feature engineering libraries such as Featuretools to further extract features. In each stage of this process, the user can leverage their expertise to tailor the generation process to the most promising, relevant features.

We conclude that SigPro provides an effective solution to signal feature extraction when simplicity and domain knowledge are critical.

[1] https://github.com/sintel-dev/Zephyr

[2] In particular, the BandMean aggregation was contributed by domain experts as motivated by empirical usage of SigPro to characterize an acute event.

4.3 Vibrations Dataset Example

Lastly, we walk through an exploratory feature engineering example using a real-world vibrations turbine dataset taken from an Iberdrola case study. [3] When concatenated, the raw dataset contains 47 columns and over 6000 rows collected from 2020 to 2022. Each row contains a timestamped series of observations by a particular sensor; in addition, the data columns provide additional readings and information about the turbine and site itself. We are primarily interested in transforming the `values` column of the dataset and will use the `rpm` column to provide the `sampling_frequency` argument as needed. For brevity, we assume that we have cleaned and collected the appropriate data files in the `vibrations.csv` file (see Appendix D).

```
1 import pandas as pd
2
3 vibrations = pd.read_csv('vibrations.csv') # Import vibrations dataset
4 vibrations['sampling_frequency'] = vibrations['rpm']
```

Figure 4-2: Importing the Vibrations Dataset.

Our sample pipeline will apply the `FFTReal` transformation to the `values`, followed by three `BandMean` aggregations, each covering a disjoint frequency range. Since we would like to apply a single sequence of transformations followed by several aggregations, the most appropriate choice is a linear pipeline (Figure 4-3).

```
1 from sigpro.basic_primitives import BandMean, FFTReal
2 from sigpro.pipeline import build_linear_pipeline
3
4 transformations = [FFTReal()]  # tag: fft_real
5 aggregations = [BandMean(200, 400).set_tag('band24'),
6                 BandMean(400, 600).set_tag('band46'),
7                 BandMean(600, 800).set_tag('band68')]
8 signal_pipeline = build_linear_pipeline(transformations, aggregations)
```

Figure 4-3: Forming the `SigPro` pipeline with `build_linear_pipeline`.

[3] https://github.com/sintel-dev/Iberdrola-case-studies

Just as in Section 4.1, the final step is to process our signal values and produce output features, which we show in Figure 4-4. We can then plot several histograms of our computed band-mean values after performing FFT, and compare their distributions (Appendix D).

```
import matplotlib.pyplot as plt

# Process the signal
processed, features = signal_pipeline.process_signal(vibrations)

for feature in features:  # Plot feature histograms
    plt.hist(processed[feature], label = feature)
plt.show()
```

Figure 4-4: Processing the signal and plotting the output features.

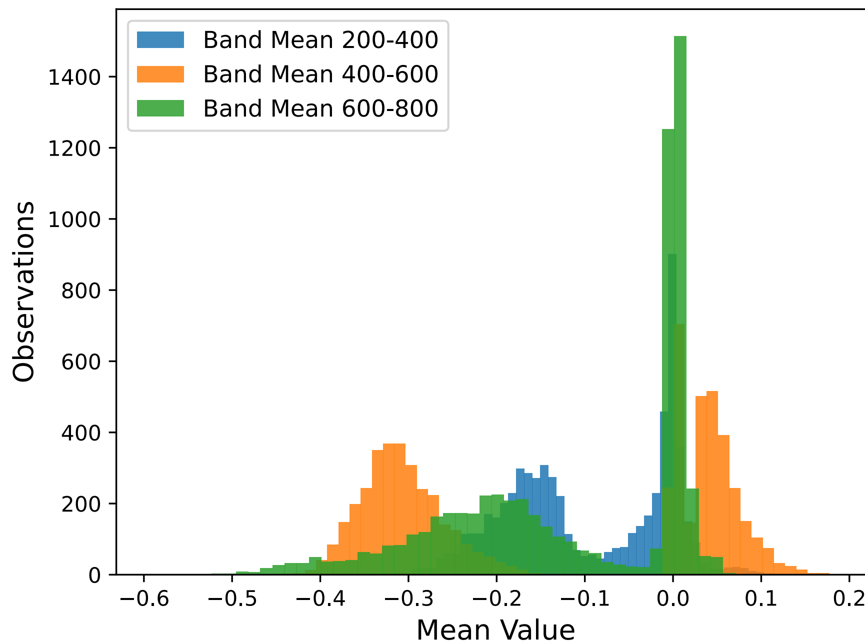


Figure 4-5: Distribution of mean (transformed) feature values across each band.

As we would in `Zephyr`, once we have generated our output features, we can either continue exploring and transforming the data with `SigPro`, or proceed with deep feature synthesis and downstream modeling.

Chapter 5

Discussion

5.1 Why SigPro?

It is useful to consider where `SigPro` stands amidst the significant body of work devoted towards automating feature engineering pipelines that can automatically generate and select features from raw data, especially as part of an end-to-end modeling procedure. As often noted, deep learning is one candidate approach for this task [10]; though deep approaches can often be successful in predictive contexts and even rival expert feature engineers at a significantly lower labor cost [11], they generally suffer from very poor interpretability [5] and often struggle in data-poor situations without careful domain-specific intervention [20]. Thus, SME participation in some form may not be avoidable. Overall, deep feature engineering aims to substitute the need for expert interaction, in contrast, `SigPro` empowers SME discretion within the feature engineering process. This proves especially valuable for *signal* data, where feature extraction patterns are often idiosyncratic to the specific application [17].

If human expertise is an essential aspect of the feature engineering process, why go so far as to build a primitive-based framework to automate it at all? It is true that existing open-source Python libraries such as `numpy`, `scipy`, and `scikit-learn` already provide a broad set of tools that can be used for feature engineering. However, such tools are less intuitive for non-developers and SMEs looking to leverage Python for their own domains; function interfaces can be quite complex with numerous optional

parameters, and modifying ad-hoc pipelines can prove quite time-consuming without significant experience with machine learning libraries.

More subtly, simply relying on several disparate third-party solutions for domain-specific tasks gives rise to significant amounts of *glue code* and the formation of thick *pipeline jungles* [16]. As Sculley et al. observe, using generic packages in ML systems increases the difficulty of leveraging domain expertise altogether. To combat these problems, Sculley et al. recommend wrapping black-box packages into standard APIs and approaching feature extraction holistically. Thus, by providing an additional layer of abstraction with a user-friendly API, `SigPro` provides its users with a simple yet flexible approach towards building bespoke signal feature engineering pipelines in their own work.

5.2 Design Rationale

As we saw previously, `SigPro` extends the primitive-pipeline model introduced in `MLBlocks` to extract signal features, but the two libraries do not treat primitives in the same manner. In particular, `MLBlocks` explicitly opts for a JSON-based approach for storing primitives [19] and wraps them with a flexible parser, as opposed to the class-subclass approach we have presented thus far. This discrepancy, of course, begs the question: why not solely adopt the JSON-parsing approach of `MLBlocks` instead?

To answer the question, we first examine the rationale for the pure JSON-based approach to primitives taken in `MLBlocks`, and their relevancy in dictating the implementation of `Primitive` objects in `SigPro`.

- **Library size.** Using a JSON specification for `MLBlocks` primitives reduced the need to maintain Python implementations for each potential subclass. However, the implementations for the underlying fitting and prediction methods must exist for any library primitive, as well as the corresponding JSON specification. An additional library, `MLPrimitives`[1], has been published for use with `MLBlocks`, providing primitive annotations and necessary Python code for integration with

[1] https://github.com/MLBazaar/MLPrimitives/

the `MLBlocks` API. Thus, the end user is still in a position to store integration code and primitive annotations even for pre-built primitives. In the case of `SigPro`, our goal of a self-contained feature engineering tool easily accessible to non-experts compels us to provide the necessary signal processing-specific primitive implementations in the library itself. We conclude that maintaining some amount of integration code/annotations for each primitive is not avoidable, and in fact desirable, in ensuring that `SigPro` remains accessible to its users.

- **Language-generality.** As pointed out in [19], the main benefit of the JSON design is that it ensures block specifications are not implemented in a Python-specific format; the `MLBlock` class interacts through its JSON annotations through a collection of parsers. This is not entirely the case for `SigPro`; while primitives do produce JSON annotations as required by the `MLPipeline` API, primitives themselves are not language-agnostic. Nevertheless, our usage of the `MLPipeline` backend with produced JSON annotations means that pipeline execution retains to some extent the language independence of `MLBlocks`. We ultimately chose to prioritize the clarity of our Python API over potential extensions to other languages.

- **Ease of modification.** Finally, Xue points out that their JSON approach allows developers to easily update the functionality of the `MLBlock` base class by updating only the class itself and modifying the parsers. We note in this scenario that due to the subclassing relationships in `SigPro`, updating the functionality of the `Primitive` base class would also require modifying only a single parent class and updating parsers/constructors as appropriate.

In addition to its flexibility, we show how `SigPro`'s modular, class-based interface supports our other design goals as articulated in Chapter 3.

- **Streamlined Development.** Developing with `SigPro` should be very straightforward, and code written with the library should be easy to understand and modify as needed. By abstracting away the need to create and parse custom

JSON annotations and simplifying the instantiation of individual primitives and pipelines, `SigPro` increases the clarity of existing feature engineering pipelines and eliminates much of the labor associated with creating new pipelines. While domain experts often prefer to write functions as opposed to classes [18], the `get_primitive_class` and `make_primitive_class` methods exploit Python's dynamic type creation so that no class definitions need be explicitly written (Figure 3-5). Therefore, users can write primitive code in the style that best matches their experience and needs.

- **Customization of Primitives and Pipelines.** As we have seen, `SigPro` explicitly supports writing and contributing custom primitives. After writing the primitive operation function itself, the contributor does so by either calling `make_primitive_class`, or creating a custom subclass for the primitive and then calling the object's `write_primitive_json()` method, to record the `MLBlocks`-compatible JSON annotation in the appropriate directory. Therefore, the contributor is not responsible for manipulating or parsing the JSON annotation.

  Moreover, pipelines themselves are simple to build and modify, and benefit from this design as well. Since all `Pipeline` objects need only to work with the `Primitive` API of their constituent primitives, they can be instantiated and used to process signals without regards to the specific properties of their components. Users can therefore freely use, substitute and contribute `Primitives` within their pipelines.

Chapter 6

Conclusion

In this thesis, we have presented `SigPro` to meet our goal of a flexible, customizable, and user-friendly library for knowledge-driven signal feature engineering. To achieve these goals, we developed a class-based library of primitives that can be efficiently composed into pipelines, enabling subject matter experts to easily incorporate their domain knowledge into the feature generation process without needing to write extensive and difficult glue code.

Our specific improvements to the `SigPro` library include:

1. **Reworking `SigPro` primitives to be class based**, thereby improving the clarity of `SigPro` usage.

2. **Revising the primitive contribution process** to leverage our new Python-ic primitive representation with a use-as-needed paradigm.

3. **Demystifying pipeline usage** by simplifying the code necessary to construct and use pipelines.

4. **Adding tree and layer pipeline construction patterns**, as facilitated by our new `Pipeline` specification.

5. **Incorporating SME feedback** into library features and design – in particular, support for primitive transformation layers within a single pipeline.

6. **Providing comprehensive documentation and demonstrations** for existing primitives and pipeline usage, speeding up the onboarding process for future first-time users.

For future editions of `SigPro`, we may be interested in expanding our built-in primitive library further to accommodate commonly used custom functionality. We could also consider supporting additional feature composition patterns and their implications for pipeline construction. In any case, further collaboration with our SME users and `Sintel` community will be greatly beneficial to ensure that `SigPro` remains an ideal signal feature engineering library.

Appendix A

Primitive Interface

Method	Parameters	Return

`Primitive`	`primitive`
	`primitive_type`
	`primitive_subtype`
	init_params
		`Primitive` object with hyperparameter values `init_params`.

`get_name`	N/A	Name (full path) of primitive function.
`get_tag`	N/A	User-given tag of primitive.
`get_inputs`	N/A	`dict` of primitive inputs.
`get_outputs`	N/A	`dict` of primitive outputs.
`get_type_subtype`	N/A	`tuple` of primitive type and primitive subtype.
`get_hyperparam_dict`	N/A	Dictionary containing hyperparameter values.
`make_primitive_json`	N/A	`dict` containing primitive JSON-style annotation.
`set_tag`	`tag`	Tagged primitive object.
`set_primitive_inputs`	`primitive_inputs`	None
`set_primitive_outputs`	`primitive_outputs`	None
`set_context_arguments`	`context_arguments`	None
`set_tunable_hyperparameters`	`tunable_hyperparameters`	None
`set_fixed_hyperparameters`	`fixed_hyperparameters`	None
`write_primitive_json`	*primitives_path* *primitives_subfolders*	Path to written primitive JSON annotation.

Table A.1: Interface of the `Primitive` class with optional parameters italicized.

Appendix B

Available Primitives

Primitive	Hyperparameters	Type	Subtype	Description
`Identity`	None	Transformation	Amplitude	Return the amplitude values as-is.
`PowerSpectrum`	None	Transformation	Amplitude	Generates the power spectrum.
`FFT`	None	Transformation	Frequency	Apply a discrete Fourier transform to the values.
`FFTReal`	None	Transformation	Frequency	Apply a discrete Fourier transform to the values and return the real parts.
`FrequencyBand`	`low` `high`	Transformation	Frequency	Filter between `low` and `high` band frequencies.
`STFT`	None	Transformation	Frequency-time	Apply a short-time Fourier transform to the values.
`STFTReal`	None	Transformation	Frequency-time	Apply a short-time Fourier transform to the values and return the real parts.
`CrestFactor`	None	Aggregation	Amplitude	Compute the ratio of the peak value to the RMS of the values.
`Kurtosis`	`fisher` `bias`	Aggregation	Amplitude	Compute the kurtosis (Fisher or Pearson) of the values.
`Mean`	None	Aggregation	Amplitude	Compute the mean of the values.
`RMS`	None	Aggregation	Amplitude	Compute the root mean square of the values.
`Skew`	None	Aggregation	Amplitude	Compute the skew of the values.
`Std`	None	Aggregation	Amplitude	Compute the standard deviation of the values.
`Var`	None	Aggregation	Amplitude	Compute the variance of the values.
`BandMean`	`min_frequency` `max_frequency`	Aggregation	Frequency	Compute the mean of values where the signal is filtered between `min_frequency` and `max_frequency`.

Table B.1: List of available primitive objects in `SigPro`.

Appendix C

Pipeline Interface

Functions	Parameters	Return
`LinearPipeline` `build_linear_pipeline`	`transformations` `aggregations`	A `LinearPipeline` generating all features from applying all transformations, then separately applying each aggregation.
`build_tree_pipeline`	`transformation_layers` `aggregation_layers`	A `LayerPipeline` generating all features in the Cartesian product of the transformation and aggregation layers.
`LayerPipeline` `build_layer_pipeline`	`primitives` `primitive_combinations`	A `LayerPipeline` generating all features specified in `primitive_combinations`.
`merge_pipelines`	`pipelines`	A pipeline generating all features produced by at least one input pipeline.

Table C.1: A list of available factory methods and constructors to manipulate pipelines.

Instance Methods	Parameters	Return
`get_primitives`	N/A	A list of primitives used in the pipeline.
`get_output_features`	N/A	A list of output features produce by the pipeline.
`get_input_args`	N/A	Input arguments to the pipeline.
`get_output_args`	N/A	Output arguments to the pipeline.
`get_pipeline`	N/A	`MLPipeline` object wrapped by the pipeline.
`process_signal`	`data` *window* *values_column_name* *time_index* *groupby_index* *keep_columns* *input_is_dataframe*	A Dataframe containing the features extracted by the pipeline from the input `data` and any non-feature columns given by `keep_columns`.

Table C.2: A list of instance methods supported by `Pipeline` objects with optional parameters italicized.

Appendix D

Vibrations Dataset Example Code

Here, we provide the full code used to produce our visualization in Figure 4-5. [1]

```
import os
import pandas as pd
import matplotlib.pyplot as plt
from parser import extract_cms_jsons
from sigpro.basic_primitives import BandMean, FFTReal
from sigpro.pipeline import build_linear_pipeline

# Build the vibrations Dataframe
data_path = os.path.join('..', '..','..', 'VibrationsDataset')
years = ['2020', '2021', '2022']
vibrations = []

for year in years:
    vibration_path = os.path.join(data_path, 'Vibration', year)
    for subfolder in os.listdir(vibration_path):
        folder = os.path.join(vibration_path, subfolder)
        if os.path.isdir(folder):
            df = extract_cms_jsons(folder)
            vibrations.append(df)

vibrations = pd.concat(vibrations)
vibrations['sampling_frequency'] = vibrations['rpm']

# Build the pipeline
transformations = [FFTReal()]
```

[1] Data I/O code adapted from https://github.com/sintel-dev/Iberdrola-case-studies/blob/main/notebooks/

```
26 aggregations = [BandMean(200, 400).set_tag('band24'),
27                 BandMean(400, 600).set_tag('band46'),
28                 BandMean(600, 800).set_tag('band68')]
29
30 signal_pipeline = build_linear_pipeline(transformations, aggregations)
31
32 processed, features = signal_pipeline.process_signal(vibrations)
33
34 labels = ['Band Mean 200 - 400',  # Nicer feature labels
35           'Band Mean 400 - 600',
36           'Band Mean 600 - 800']
37
38 for i, feature in enumerate(features):
39     plt.hist(processed[feature],
40              label = labels[i],
41              bins = 50,
42              alpha = 0.8)
43
44 # Formatting plot
45 plt.xlabel('Mean Value', fontsize = 13)
46 plt.ylabel('Observations', fontsize = 13)
47 plt.legend(fontsize = 11)
48 plt.show()
```

Bibliography

[1] 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.

[2] Marília Barandas, Duarte Folgado, Letícia Fernandes, Sara Santos, Mariana Abreu, Patrícia Bota, Hui Liu, Tanja Schultz, and Hugo Gamboa. Tsfel: Time series feature extraction library. *SoftwareX*, 11:100456, 2020.

[3] Richard G Baraniuk. More is less: Signal processing and the data deluge. *Science*, 331(6018):717–719, 2011.

[4] Nigel Bosch et al. Automl feature engineering for student modeling yields high accuracy, but limited interpretability. *Journal of Educational Data Mining*, 13(2):55–79, 2021.

[5] Supriyo Chakraborty, Richard Tomsett, Ramya Raghavendra, Daniel Harborne, Moustafa Alzantot, Federico Cerutti, Mani Srivastava, Alun Preece, Simon Julier, Raghuveer M Rao, et al. Interpretability of deep learning models: A survey of results. In *2017 IEEE smartworld, ubiquitous intelligence & computing, advanced & trusted computed, scalable computing & communications, cloud & big data computing, Internet of people and smart city innovation (smartworld/SCALCOM/UIC/ATC/CBDcom/IOP/SCI)*, pages 1–6. IEEE, 2017.

[6] Girish Chandrashekar and Ferat Sahin. A survey on feature selection methods. *Computers & Electrical Engineering*, 40(1):16–28, 2014.

[7] Maximilian Christ, Nils Braun, Julius Neuffer, and Andreas W Kempa-Liehr. Time series feature extraction on basis of scalable hypothesis tests (tsfresh–a python package). *Neurocomputing*, 307:72–77, 2018.

[8] Bryan Omar Collazo Santiago. *Machine learning blocks*. PhD thesis, Massachusetts Institute of Technology, 2015.

[9] Frances R Hartwell. *Zephyr: a Data-Centric Framework for Predictive Maintenance of Wind Turbines*. PhD thesis, Massachusetts Institute of Technology, 2023.

[10] Xin He, Kaiyong Zhao, and Xiaowen Chu. Automl: A survey of the state-of-the-art. *Knowledge-Based Systems*, 212:106622, 2021.

[11] 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)*, pages 1–10. IEEE, 2015.

[12] Gilad Katz, Eui Chul Richard Shin, and Dawn Song. Explorekit: Automatic feature generation and selection. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 979–984. IEEE, 2016.

[13] Yanrui Li and Chunjie Yang. Domain knowledge based explainable feature construction method and its application in ironmaking process. *Engineering Applications of Artificial Intelligence*, 100:104197, 2021.

[14] Hiroshi Motoda and Huan Liu. Feature selection, extraction and construction. *Communication of IICM (Institute of Information and Computing Machinery, Taiwan)*, 5(67-72):2, 2002.

[15] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

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

[17] Garima Sharma, Kartikeyan Umapathy, and Sridhar Krishnan. Trends in audio signal feature extraction methods. *Applied Acoustics*, 158:107020, 2020.

[18] Micah J Smith, Carles Sala, James Max Kanter, and Kalyan Veeramachaneni. The machine learning bazaar: Harnessing the ml ecosystem for effective system development. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 785–800, 2020.

[19] William Xue et al. *A flexible framework for composing end to end machine learning pipelines*. PhD thesis, Massachusetts Institute of Technology, 2018.

[20] Xiaofeng Zhang, Zhangyang Wang, Dong Liu, Qifeng Lin, and Qing Ling. Deep adversarial data augmentation for extremely low data regimes. *IEEE Transactions on Circuits and Systems for Video Technology*, 31(1):15–28, 2020.