SenseML: A Platform for Constructing IOT Data Pipelines

by

Donghyun Michael Choi

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

September 2017

© Massachusetts Institute of Technology 2017. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
September 11, 2017

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

Accepted by . . . .
Christopher Terman
Chairman, Department Masters of Engineering Thesis Committee

SenseML: A Platform for Constructing IOT Data Pipelines

by

Donghyun Michael Choi

Submitted to the Department of Electrical Engineering and Computer Science
on September 11, 2017, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, we present SenseML. SenseML is a general-purpose platform that enables users to transform sensor data from the IOT domain into a machine learning-ready format – what we call an attribute time series. It is a cloud-based platform that can process signals using user-specified functions. It offers users immense flexibility in integrating functions for transforming the data, while also providing parallel execution as a service. In addition, we enable users to contribute to the framework by submitting domain-specific signal processing functions. Such contributions are integrated into the platform and are then part of the library, available for others to use. We used the platform to generate 19 attribute time series for 9655 urban sound signals. To generate these time series, the platform did 120 million computations in approximately 140 minutes.

Thesis Supervisor: Kalyan Veeramachaneni
Title: Principal Research Scientist

Acknowledgments

First and foremost, I would like to sincerely thank my supervisor, Kalyan Veeramachaneni, for the continuous guidance and support throughout the project. Kalyan's expertise in the field served as a pivotal resource in driving the project forward. Without his dedication and involvement, this paper would not have been possible.

I'd also like to express gratitude for my fellow DAI lab members, Alec Anderson and Bennett Cypher, for the accompaniment over the summer and providing a very pleasant work environment.

I am also grateful for Pavika Buddhari for the continuous support spiritually throughout this thesis writing process, as well as in life. Thanks for never failing to encourage me during harder times.

I would also like to acknowledge Kevin Lu. I believe he knows why.

Finally, I would like to thank my family and friends for the unconditional support throughout the years in all aspects of my life. Thank you to those who have been part of this journey in some way, no matter how small, as I would not have been able to accomplish this without you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As the Internet of Things (IOT) plays a growing role in our society, the amount of data that must be processed and analyzed in order to keep things running smoothly is constantly increasing. It is vital that the many industries in the IOT landscape be able to efficiently analyze the data produced by their devices. They need to be able to use this data to – on the one hand – improve settings for more everyday use, and on the other to predict adverse events like looming machinery failure. In all cases, developing these solutions requires machine learning.

Most of the datasets that businesses within these industries are working with are made up of large, highly granular time series data collected using sensors. Generating predictive and machine learning models from this type of data has been the focus of both the academic and industrial communities the past two decades, if not longer.

Recent times have brought us a plethora of machine learning platforms meant to help developers build, test, and deploy machine learning solutions. However, because they aim to be general purpose, most of these platforms can only work with certain representations of data. If a data scientist tries to use these platforms to work with the aforementioned IOT data – which typically comes in the form of signals – they will find that they have to process the data using signal processing functions before they can feed it into the machine learning platform.

In this thesis, we present SenseML, a platform for performing signal processing on IOT data and preparing that data for machine learning platforms. The name

SenseML represents the purpose of the platform – bridging the representational gap between signals generated from sensors (Sense) and the input data representation required by the machine learning platforms (ML).

It is daunting to build a general-purpose platform that will work for all IOT-related problems. However, a few foundational insights made this work possible:

- Even when they come from many disparate domains, signals always fall into certain categories, and can be characterized as such. These categories are not domain-specific, and can be used to create processing functionality. We discuss a few of these categories in Chapter 2

- When we examined the process data scientists use to transform raw signal data into a machine learning-ready time series (a.k.a. a featurized format), we noticed that everyone followed the same steps we list in Chapter 3. Regardless of the source of the data – humans, turbines, cars, engines or even mobile phones – the processing steps remain the same: they were merely named differently. This allowed us to structure the process and develop a general-purpose platform.

- While we can create the structured process and stages that the data has to go through, and enumerate the possibilities based on signal types, the specific processing functions can vary significantly from domain to domain, or from one signal to another. However, the signatures of how these functions are written (inputs and outputs) are somewhat standard, allowing us to create abstractions that make the platform extendable.

This thesis is organized into several chapters. In chapter 2, we give an overview of the different types of signals that are commonly encountered in a domain. Chapter 3 outlines the steps involved in processing raw signal files, as observed across many domains and applications. In Chapter 4, we introduce SenseML and its underlying architecture. In Chapter 5, we introduce `SenseML Client`, a simple Python interface built to run SenseML's signal preprocessing steps. In Chapter 6, we demonstrate the tool on a real world dataset. Finally, in Chapter 7, we conclude by providing a list of future work that could be done to improve SenseML.

Chapter 2

Heterogeneity of Signal Sources

To develop a system that works for all possible IOT signals, we consider the multiple heterogeneities involved: the differences in signal types, what domains they come from, their possible uses, and the multitude of ways in which they are stored. In this chapter, we present a review of different dimensions over which signals could be characterized. In later chapters, we design solutions that address this heterogeneity.

2.1 Signal Types

There are certain salient properties of signals that allow them to be categorized. The pre-processing and analyzing steps that must be performed on these signal vary depending on its category.

2.1.1 Regular vs Irregular Sampling

A "regular" signal is one whose samples are recorded at fixed time intervals, and which produces a regularly-spaced time series. Most signals that are obtained from sensors can be expected to be regular, as sensors are programmed to take samples at a pre-specified rate.

When a signal is irregular, its samples are not taken at fixed time intervals. Such signals are often event-driven. For example, a signal that creates a battery-low alert is only generated when the battery is low.

2.1.2 Periodic vs Aperiodic

A periodic signal has a pattern that repeats over a measurable time frame, often produced by some underlying mechanism that has a periodicity. In electrical and mechanical systems, most machines operate with rotational or cycle-based machinery. Wind turbines rotate with a certain rpm; CPUs process information at a fixed clock cycle per second. Thus, signals measured from these machines will be expected to be periodic. In the case of physiological systems (e.g. electrocardiography), the periods of these measurements often correlate with the heart rate.

Aperiodic signals, on the other hand, do not have a repeating pattern. This category of signals encompasses the vast majority of sound recordings, such as acoustic and surround sounds, as well as temperature measurements inside a building.

2.2 Domains

Most of the signal data that ends up being processed and analyzed is recorded using sensors. Depending on the eventual application and the system being monitored, different types of sensors are used for this recording. For example, when machinery is being monitored, the most common types of signals recorded include:

- temperature
- pressure
- vibration
- fluid-property
- positional orientation

Other domains include: monitoring computing devices that measure usage of different resources within a machine (e.g. CPU or memory), monitoring human physiology (in which case sensors measure different physiological aspects like heart rate or blood pressure), and activity monitoring (which captures our use of complex systems, such as temperature and light settings in a home, sounds, and others). Often it is important to identify what domain the signal is coming from, as this may influence how it is processed. For example, a temperature signal is processed differently when it is coming from a home, versus when it is coming from the high-pressure compressor section of a turbine.

2.3 Applications

Industries that utilize sensor data are seeking two main objectives: event prediction and event classification.

2.3.1 Event Prediction

The main goal in many industries is predict an outcome ahead of time. For example, in the equipment industry, the goal is to maintain the condition of a particular piece of machinery. In this context, the problem of interest is to predict if and when a machine will experience a failure event, given a log of sensor data.

One applied example comes from wind turbine systems. Wind turbines are typically installed with sensors that measure vibrations, temperature, pressure, and position. Over the past few decades, there has been growing interest in using this data for damage detection in their propellers. [15] This type of condition monitoring and failure detection is also sought after in flight control systems [16], hard drive maintenance [3], and physiological systems [21].

2.3.2 Event Classification

Another emerging objective is to recognize events of interest in a signal time series. Environmental sound recognition and classification is a growing area of research, with applications in multimedia retrieval and urban informatics [17]. In this application, the challenge is to detect what objects or activities are creating the sounds, given the signal data.

2.4 File Formats

One of the challenges inherent in building a general-purpose system is the number of disparate file formats that signals can be stored on, and thus could be presented to the system. Different industries tend to have different specifications that work well for their applications. Additionally, some file formats tend to contain more than just raw data samples, and include annotations that may make processing simpler and more efficient [14].

We can examine some file formats commonly seen in different industries:

- For multi-channel physiological data, the widely-accepted file format standard is the European Data Format (EDF) [14].
- Sound files tend to be stored in wav or mp3 formats.
- The comma-separated values (CSV) format is a simple format that allows data to be used in multi-application contexts.

For each of these file formats, tools are available to extract raw data samples.

Chapter 3

From Raw Signals to Machine Learning-Ready

For most applications, signal data obtained from sensors comes in the form of high-resolution, often high-frequency time series data. In order to apply machine learning techniques to this raw signal time series, it is first necessary to pre-process the data. One of the main problems with using raw signal time series data is that the dataset is often too large to efficiently process in a reasonable time frame. This enormous size is a function of the frequency at which the data is sampled, also known as the "sampling frequency." For instance, most sound files have a sampling frequency of 44.1kHz or greater, and ECG signals are recommended to be sampled at a rate of at least 200-500Hz [12]. Higher-frequency sampling, if processed carefully, can yield more accurate results, so it is desirable to sample at a high frequency when possible. However, machine learning algorithms, including time series modeling software packages, expect a higher-level signal then what the sensors can produce.

The goal of the preprocessing steps is to reduce this dimensionality by (a) splitting the signal data into segments, and (b) extracting any attributes that may indicate the segment's unique properties. More specifically, the steps involved are as follows:

- Decoding
- Segmenting

- Validation
- Attribute Extraction

We found this to be a standardized process across many domains, regardless of the signal type and end goal.

3.1 Decoding

"Decoding" is the process of taking the input signal data file and extracting the raw sample data. For most common file formats, numerous tools and scripts exist to enable this. As an example, SoX ([5]) is an open-source audio editing software that is able to read raw sample data for a myriad of file formats. The PhysioNet Waveform db software package ([6]) provides a way to read those file formats that are commonly used for physiological signals (e.g. EDF).

Another component of decoding input data file into raw samples is filtering. Filtering is the process through which raw signal data is "cleaned up" to remove any potential noise. Finding the correct filter heavily depends on the domain and the problem being solved. One of the most commonly used filters is a low-pass filter, which removes any extraneous high-frequency noise from the signal. The problem with defaulting to this choice is that sometimes, this higher-frequency data can be indicative of a particular event that we are interested in predicting. For instance, within hardware performance data, random high-frequency spurts can indicate an imminent hardware failure. Therefore, if data scientists are given pre-processed, low-pass filtered data in this context, they would be analyzing data that is missing one of the markers of hardware failure.

In addition to the commonly-used, generic filters, domain- and problem-specific filters may also be applied. In human speech analysis, "liftering" functions are used to pinpoint different characteristics of the signal data ([13]).

3.2 Segmenting

The main goal of segmentation is to split a signal into independent segments for analysis. For periodic signals, this task is fairly straightforward. First, we make a distinction between two fundamentally different signal type: signals produced by rotating machinery, and signals captured from the human body. As most machines operate with an underlying fixed mechanism that has some periodicity – for example, 350 revolutions per minute – machine signals have a period of consistent duration. We can decide to split the signal on fixed intervals based on this period and the sampling frequency.

For human body signals, we can no longer assume a uniform duration for each period, and we must expect a lot more variety and irregularity. As we know, our heart beats at different rates based on our activity. Thus, the task of splitting the signals into their periodic segments is no longer as trivial, and requires more sophisticated methods. For example, there exists an onset detection algorithm to detect and separate individual heartbeats within ECG readings. The PhysioNet waveform dB software package provides a commonly used implementation of such an algorithm.([6])

For aperiodic signals, proper segmenting methods can vary quite a bit depending upon the domain and the use case. For example, a common approach for acoustic event detection is to use overlapping window "frames" to define the segments. The values for the segment length and the percent overlap between segments are fixed, and normally fall under a specific range that has been agreed upon in the field. ([7],[10],[8])

3.3 Validation

After segmentation, we verify the validity of each segment. This step removes any malformed segments that may affect the correctness of the resulting dataset after attributes are extracted. Once again, the definition of validity varies depending on the domain and the problem of interest. For ABP waveforms, one effective algorithm for detecting abnormal beats is the Signal Abnormality Index (SAI) algorithm ([19]).

3.3.1 Attribute Extraction

After segmenting the signal data, we can extract attributes for each of the segments. The attributes we may be interested in fall under multiple categories.

- **Statistical**: This first category covers the statistical functions applied to the data in the segment (e.g. min, max, average).
- **Domain-specific**: Other attributes are domain-specific. For example, in ECG measurements, the distances between the QRST peaks in the signal are strong indicators of the health of the heart.
- **Time domain**: These attributes capture properties of the signals in the time domain, such as RMS value and crest factor.
- **Frequency domain**: These attributes capture properties of the signals in the frequency domain. A transform is applied, and the attributes are calculated on the transformed version of the signal. Examples include the spectral energy present in a specific band.
- **Intra-segment**: Attributes within this last category capture the relationship between data samples across multiple consecutive segments.

3.4 Two End-to-End Examples of Signal Processing Pipelines

In this section, we will outline the signal preprocessing steps taken for two applications.

3.4.1 Physiological Signals

Electrocardiography (ECG):

- Decoding: Parsing the EDF(+) file into raw sample data, using PhysioNet wfdb software.

- Segmentation: Applying an onset detection algorithm to detect measurements from individual heartbeats.
- Validation: Checking each heartbeat for validity.
- Attribute Extraction: Extracting relevant attributes, including crest, diastole, kurtosis, pressure area, skewness, systole duration, and pulse.

3.4.2 Acoustic

Urban Sounds Classification:

- Decoding: Parsing wav, and mp3 files using SoX (or other software tools).
- Segmentation: Segmentation using a 40ms segment length with 50% segment overlap.
- Validation: There is no need for validation.
- Attribute Extraction: Extracting spectral features (e.g. ZCR, Energy, and Entropy) and MFCC.

Chapter 4

SenseML

SenseML aims to be a cloud-based platform for processing IOT signals and preparing them for machine learning, a step that currently takes up most of a data scientist's time during a predictive modeling task. Its goal is to provide:

– an easy-to-use parallel computing infrastructure for processing massive amounts of signals.

– a set of simple, cloud-based interfaces that allow users to perform the aforementioned preprocessing steps, and

– a set of simple interfaces that allow domain experts and other users to integrate their domain-specific processing functions.

When given a large repository of sensor data, SenseML outputs a low-frequency, "compressed" attribute time series that is ready for machine learning. Users can integrate this time series with a variety of machine learning approaches, including `state-space models` and `classification approaches`. In this chapter, we describe the parallel architecture that underlies SenseML.

4.1 Motivation

We begin by recognizing that although many of the signal processing tasks presented in the previous chapter can be executed in parallel, no system currently exists to enable this. Additionally, we note that the intermediate representation of regular, low-frequency time series preserves most of the information contained in the raw signal, and is the most amenable representation for machine learning. Thus, processing these signals and storing these pre-computed representations proves extremely beneficial in scaling the machine learning studies that follow. The parallelization architecture in SenseML is motivated by four main observations:

1. It is computationally expensive to process raw signal data samples, because the data is often sampled at high frequencies. As a result, data scientists either limit the scope of their studies to a subset of the data, or perform processing that targets the problem at hand. For example, if a data scientist wants to predict an acute-hypotensive episode, defined in a specific way, they will find patients who have experienced this event, along with an equivalent number who did not, and process these signals. They never revisit this step, even though it may be beneficial to explore a different set of patients as a control, or to take another look when the definition of the AHE changes.

2. In some applications, the choice of computational functions used for pre-processing steps appears to be ad hoc, with little to no explanation as to why they were chosen. A system that can execute steps quickly allows for the exploration of multiple options available for these steps.

3. Although data scientists may decide to build a distributed system themselves, it is non-trivial to build such a system for the preprocessing steps.

4. On the other hand, it is possible to build a general-purpose framework with well-defined abstractions that supports parallelization of these steps across many different domains within IOT, as we will show in this thesis.

SenseML utilizes Amazon Web Services ([1]) to create a distributed cloud system designed to run the signal data preprocessing steps efficiently. As these steps are inherently parallelizeable for different signal files, users get huge performance benefits from using a distributed architecture. SenseML also provides a simple Python interface for our distributed preprocessing service.

Finally, SenseML provides a global repository of computational functions that can be applied to IOT data. This allows people to browse these functions and explore their hyperparameters, as well as enhance the repository with their own contributions.

4.2 Previous Work

Sense-ML was adapted from a similar project called PhysioMiner [11]. PhysioMiner is a system designed for processing and analyzing physiological waveforms. Because the underlying steps of segmentation and attribute extraction are similar between the two systems, they have many design similarities. In particular, Sense-ML uses the same cloud architecture and has the same module types as PhysioMiner.

Sense-ML differs from PhysioMiner in the following ways:

– It generalizes the system, allowing it to be used over the entire IOT domain. To accomplish this, we had to develop mechanisms that allowed for the ingestion of many different signal file types and formats.

– It provides a much broader library of processing functions to account for data from different domains, including irregular and aperiodic signals.

– It finalizes the function-defining abstractions for each of the steps, enabling users to write custom functions and integrate them with the framework.

– It allows users to integrate their own signal processing functions and contribute to a repository of these functions.

4.3 System Architecture

Figure 4-1 outlines the overall system architecture of SenseML.

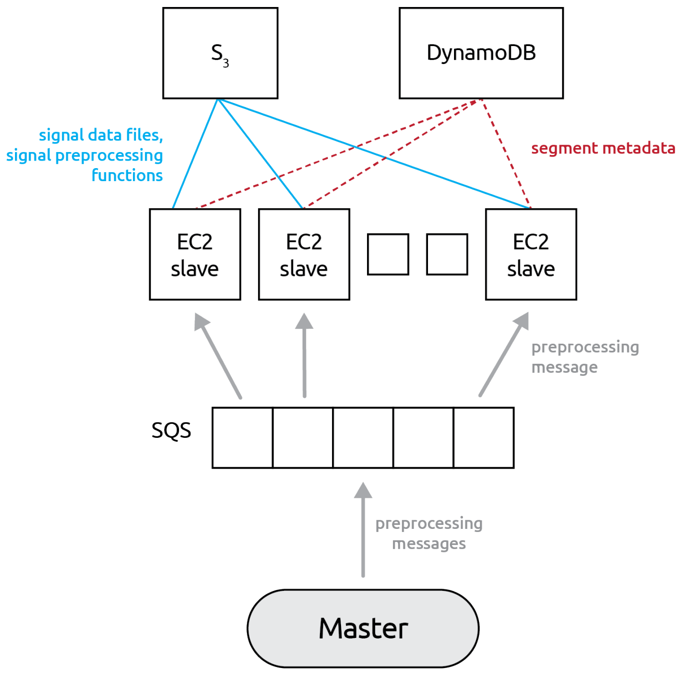

Figure 4-1: The basic system architecture for SenseML. There are three main components and their interactions - database, master-slave framework, and file storage

4.3.1 File Storage

The SenseML system's main storage component uses an Amazon S3 bucket. In addition to storing user-provided data files, the S3 bucket also maintains the software repository of pre-processing functions, and the SenseML jar.

To be able to run SenseML, the data and software in the S3 bucket must follow a specific directory structure. We show the directory structure below:

```
data/
    |-- entity_id
            |-- file1
            |-- file2
    |-- entity_id2
        ...
repository/
    |-- attributes/
    |-- signal-decoders/
```

```
        |-- segmenters/
    jars/
        |-- main.jar
```

Here are the key highlights of this directory structure:

- Importantly, all signal files must be stored under the `data/` directory. In this folder, each entity-instance gets its own subfolder, which stores all the signal files and meta-files associated with that entity-instance. For example, consider a dataset from wind turbines. The data from `Turbine8984` (entity-instance) will be stored in a folder called `Turbine8984/`.

- All the software with functions for different steps of the signal processing should be stored in the `repository/` folder. If the user clones a SenseML repository from `github`, s/he will have this folder readily available.

- The jars should be stored in a folder called `jars/`.

4.3.2 Database To Store Results

SenseML uses a NoSQL database called DynamoDB to store the output attribute time series. The data is organized into a schema consisting of two tables: `signals` and `attributes`.

The `signals` table will store information regarding the signal files. This includes a randomly generated, unique `signal_id`, the `file_path` of the data file as stored in the S3 bucket, and the `entity_id` to which the signal belongs. Note that multiple signals can belong to the same `entity_id`. Table 4.1 shows an example of a valid `signals` table.

entity_id	**signal_id**	**file_path**
203962	1TeA5Py	data/203962/203962-5-0-1.wav
203962	ZDMSW3t	data/203962/203962-5-0-0.wav

Table 4.1: Signals Table

The `attributes` table will store the time-series data that results from processing the signal file. Each row in this table represents a segment and contains a column for each of the corresponding attributes, along with metadata extracted from that segment. Table 4.2 shows an example of a valid attributes table.

signal_id	**segment_id**	**interval_start**	**duration**	**frequency**	**energy**	...	**valid**
RB8dlvt	0	0	556	44100	1437.0589	...	true
RB8dlvt	1	278	556	44100	1448.2890	...	true
RB8dlvt	2	556	556	44100	1672.176	...	true

Table 4.2: Attributes Table

4.3.3 Master/Slave Architecture For Distributing Compute

SenseML is built upon a master-slave architecture that uses Amazon SQS as its global message queuing system. The different signal preprocessing steps (signal decoding, segmentation, attribute extraction) are separated into parallelizable tasks, which are then encoded as messages to this system. The master node is responsible for populating the message queue with these tasks.

Slave nodes, on the other hand, are responsible for completing the tasks in the message queue. A slave node will continuously query the queue for available messages, decode each message, and carry out the encoded task. After completing the task, the slave will update the database with the result.

SenseML uses Amazon EC2 instances for slaves. These EC2 instances are created using an Amazon Machine Image (AMI), which is a snapshot of an EC2 instance. AMIs can be used to initialize multiple EC2 instances in the same state. We provide a public AMI that contains the necessary software to run SenseML.

4.4 SenseML Modules

Sense-ML contains two modules responsible for completing the signal data preprocessing steps: Population and Attribute Extraction.

4.4.1 Population Module

The main objective of the population module is to segment the signal data files and populate the database with information about the segments. Information that gets populated in the `attributes` table includes:`sample frequency`, `start time`,`interval`, `validity` and `duration`. At the same time, the `signals` table is also populated with meta-information pertaining to the signal, if it does not already exist.

SenseML uses a segmenter function selected by the user to parse the data files.

Master

The master node first retrieves the list of data files from the S3 buckets. It then creates a `PopulatorMessage` for each data file and pushes it to the SQS.

A `PopulatorMessage` consists of a DynamoDB Table prefix, the name of the S3 bucket, the filepath of the signal data, and the filepath and parameters of the chosen segmenter function.

Slave

Upon receiving a `PopulatorMessage`, a slave node will:

1. download the specified signal data file and segmenter function from S3,
2. run the segmenter function on the signal data file with the provided function parameters, if applicable, thus creating a file, and
3. parse the resulting segments and store them into the specified DynamoDB table.

4.4.2 Attribute Extraction Module

In this module, SenseML will take the resulting segments from the Population stage and apply attribute extraction functions to each. The resulting attributes are stored in the database. To run this module, users must provide both a signal decoder function and a list of attribute extractors.

Master

The master node will scan the DynamoDb tables created during the Population stage. For each of the segments in the system, the master node creates a `AttributeExtractorMessage` and pushes it to the SQS.

An `AttributeExtractorMessage` consists of a DynamoDb Table prefix, the name of the S3 bucket, the filepath of the signal data, and the filepath and parameters of the signal decoder and the attribute extractor functions chosen.

Slave

Upon receiving a `AttributeExtractorMessage`, a slave node will

1. download the specified signal data file and signal decoder function from S3,
2. use a signal decoder to read raw data samples from the data file,
3. segment the data samples based on the metadata stored in DynamoDb,
4. extract each of the attributes from the segments, and
5. update the segments into DynamoDB with the extracted attributes.

Chapter 5

SenseML API

Usability is one of our key objectives, and drives the design of SenseML. We expect three types of users, which we describe briefly below:

– **Developers/Data scientists**: These users will turn to SenseML to solve data problems. They pick functions from the repository for different processing stages, and would prefer to run SenseML on a high-level `API`.

– **Domain experts**: These users have the same goals as developers and data scientists, but would like to modify the processing functions or integrate their own, in order to make them as specific as possible to their domain. They may or may not contribute to the repository.

– **Contributors**: These are users who will contribute to the repository by submitting to the library of processing functions.

To address the first and second user type, we developed a Python client, called `SenseMLClient`. `SenseMLClient` interfaces with the Java archive that handles the AWS cloud system, and runs the populator and attribute extractor modules.

5.1 SenseML Client

Users employ SenseML Client to run the different stages of SenseML by following three simple steps:

– Set the configuration and initialize the client

– Upload the data and the functions repository

– Run the SenseML stages

5.1.1 Set Configuration and Initialize Client

A user must set up different settings in order to initialize the SenseML client. (It is assumed that this user has an Amazon account and corresponding credentials.)
Project configurations: There are a few project-related configurations that must be specified for the client:

- `senseml_jar`: The relative path of the SenseML jar. The Python client needs to know the location of the SenseML jar in order to run the AWS system.

- `functions_dir`: The relative path of functions repository. SenseML uses this to validate selected functions and user input.

Note that these parameters are included in order to allow users to run the client without restricting the usage to a specific directory.
AWS Credentials: Users must have an AWS account with valid credentials, along with access to the S3, DynamoDb, SQS, and EC2 services. As an additional prerequisite, users must install the AWS-CLI ([2]) with their credentials. The following is a list of AWS parameters required for initializing a `SenseMLClient`:

- `aws_key_name`: Amazon AWS access key name

- `aws_service_region`: AWS Service Region

- `ec2_iam_profile`: Name of IamProfile to use for EC2 instances (must have access to DynamoDB, S3, and SQS)

- `s3_bucket`: Name of S3 bucket used for storing data, functions, and other project files

- `dynamodb_name_prefix`: DynamoDB Table Name Prefix
- `sqs_name_prefix`: SQS Name Prefix
- `ec2_num_instances`: Number of EC2 worker instances to create
- `ec2_instance_type`: Amazon EC2 Instance type to launch, (options: r3.large, r3.xlarge, r3.2xlarge, r3.4xlarge, r3.8xlarge, c3.large, c3.xlarge, c3.2xlarge, c3.4xlarge, c3.8xlarge)
- `ec2_instance_ids`: List of EC2 instances ids to be reused
- `ec2_private_key`: Private key file for connecting to EC2 instances

```python
from SenseMLClient import SenseMLClient

S = SenseMLClient(
    aws_key_name = "senseml-demo",
    aws_service_region = 'us-east-1',
    ec2_iam_profile = "senseml-test-role",
    s3_bucket = 'senseml-demo',
    dynamodb_name_prefix = 'senseml',
    sqs_name_prefix = 'senseml',
    ec2_num_instances = 2,
    ec2_instance_type = 'r4.large',
    senseml_jar = '../repository/jars/main.jar',
    functions_dir = '../repository/'
)
```

Listing 5.1: SenseMLClient initialization example

Listing 5.1 gives an example of how to initialize a new `SenseMLClient`. Upon initialization, the client will upload the Sense-ML jar to the S3 bucket specified in preparation for the upcoming modules.

5.1.2 Upload the Data and Functions Repository

SenseML provides a repository of functions from which users can select their signal decoders, segmenters, and attribute extractors. The directory structure of the repository is as follows:

```
repository
    |-- attributes
            |-- meta.json
            |-- f1.py
            ...
    |-- signal-decoders
            |-- meta.json
            |-- f2.py
            ...
    |-- segmenters
            |-- meta.json
            |-- f3.py
            ...
```

As seen above, for each function type, there is also a meta.json file that contains additional information about the function. The details of the meta.json file are discussed in section 5.2.4. In order to run any of the SenseML modules, users must choose corresponding functions from this repository that work with the data they are operating on.

Additionally, the functions repository must be synced to the Amazon S3 bucket specified in the client configurations. This will allow remote SenseML workers to query the S3 bucket and obtain the selected functions.

`SenseMLClient` provides a function for this. For example, if the repository were accessible via the relative path `"../repository/"`, users could simply call

```
S.uploadFunctionsRepository("../repository/")
```

in order to upload the directory into the S3 bucket specified in the client configuration.

5.1.3 Running SenseML stages

Population Stage: `SenseMLClient` provides the following API for running the population stage.

```
S.populate(segmenter = (<selected_segmenter>, <hyperparams>),
            init_tables = [True|False])
```

This function will segment any uploaded data files using the specified segmenting function, and populate the DynamoDB tables. The `init_tables` argument indicates whether the DynamoDb tables should be initialized at the beginning of the run.

In order to populate the database, users must select a way to segment their raw data files. They can do this by searching through the functions repository for any relevant segmentation algorithms. They must also provide any additional parameters that the function states as arguments.

For example, consider the following truncated "audio.py" segmentation function:

```
import scipy.io.wavfile as wav, os

def segment(f, window_size, percent_overlap):
    """
    params:
        window_size = size of window in milliseconds, float
        percent_overlap = amount of frame overlap for segments,
   float
    """
    ...
    return samp_freq, start_intervals, segments
```

Listing 5.2: Segmenter example: audio.py

This segmenter function requires that the arguments '`window_size`' and '`percent_overlap`' be supplied as hyperparameters. (Note: '`f`' is simply the input file required for all segmentation functions). Thus, if a user wants to use this segmenter function, he or she must also supply a dictionary with values for each of these function arguments.

Listing 5.3 shows an example call of `populate()`, where valid hyperparameters are supplied for the "audio.py" segmenter.

```
S.populate(segmenter = ["audio.py", {"windows_size": 23.2, "
    percent_overlap": 0.5}])
```

Listing 5.3: Running the Populate Stage with the "audio.py" segmenter with hyperparameters supplied

Attribute Extraction Stage: The API for running the attribute extraction stage is

```
S.extract_attributes(
    signal_decoder = (<selected_signal_decoder>, <hyperparams>),
    attributes = [
        (<selected_attribute>, <hyperparams>),
        ...
    ]
)
```

For the attribute extraction stage, users must select a signal decoder, along with the list of attributes they wish to extract from each of the segments obtained in the population stage. The specifications for these functions are the same as those for segmenter functions – users are required to include any hyperparameters that the functions require.

Upon completion of the attribute extraction stage, the user's DynamoDb tables should consist of attribute timeseries for each of the signal files provided.

5.2 Creating your own functions

SenseML allows domain experts and contributors to to create their own signal decoder, segmenter, and attribute extractor functions. They can use such functions to process their own data, and/or to contribute to the global library. Since we are attempting to build a general-purpose framework for IOT with varied signal types and application domains, enabling contributions and accumulating a global repository of signal processing functions is paramount for the success of this framework. When we started this project, we only had attribute extraction functions for `ecg` and `abp` – both physiological signals. We wanted to process acoustic signals, and found the an implementation of corresponding attribute extraction functions at ([4]).

While designing the abstractions for these functions, we made sure that they resembled as closely as possible the processing functions written by domain experts *without* SenseML. This maximizes the chances of contribution, because it means that these experts can integrate their functions into SenseML without performing any additional work. In the following subsections, we present these abstractions, which result in the specifications for different processing functions.

5.2.1 Signal Decoder

A valid signal decoder must have the following function signature:

```python
def decode_signal(f):
    ...
    return data
```

Listing 5.4: Signal Decoder function template

where

- `f` is the input data filename
- `data` is a 2D array with dimensions `num_samples` × `num_channels`

5.2.2 Segmenter

A valid segmenter must have the following function signature:

```python
def segment(f (, hyperparams, ...)):
    ...
    return samp_freq, start_intervals, segments, num_channels
```

Listing 5.5: Segmenter function template

where

- `f` is the input data filename
- `hyperparams` are optional hyperparameters that must be provided
- `samp_freq` is the sample frequency of the data file
- `segment` is a list of 2D arrays with dimensions `num_samples_in_segment` × `num_channels`
- `start_intervals` is the corresponding start time for each of the segments
- `num_channels` is the number of channels found in the data file

5.2.3 Attribute Extractor

A valid attribute extractor must have the following function signature

```
def extract_attribute(data (, hyperparams, ...)):
    ...
    return attribute
```

Listing 5.6: Attribute Extractor function template

where

- `data` is a tuple of 2D arrays
- `hyperparams` are optional hyperparameters that must be provided
- `attribute` can be a single numeric value or a tuple/list/array of values, as specified in the meta.json

5.2.4 Metadata

When creating a new function, it is important to create a new metadata entry in the corresponding meta.json file. Metadata contains information about the function, and is used by SenseML to assess how to process that function. Currently, the metadata file for the attributes functions is the only one actively processed, as there has been no operational necessity for the segmenter and signal decoder metadata.

Metadata for an attribute extractor will consist of a JSON object with the following parameters:

- domain: A keyword indicating which types of application the attribute might prove useful to.
- description: A short description of the attribute.
- hyperparams: A list of hyperparameters required by the function.
- nChannels: The number of channels the function expects for the input data.
- nConsecutive: Information regarding whether the attribute operates on a single segment, or a list of n consecutive segments.
- includeFS: A Boolean that indicates whether the function expects the sample frequency to be provided as a parameter.
- output: Contains information about the number and naming of the outputs.

Listing 5.7 shows an example of a metadata json provided for a function that extracts the spectral flux attribute.

```
{
    "spectral_flux.py": {
        "domain": "audio",
        "description": "Computes the spectral flux feature of the
current frame",
        "hyperparams": [],
        "nChannels": 2,
        "nConsecutive": {
            "n": 2,
            "position": "left"
        },
        "includeFS": false,
        "output": {
            "type": "fixed",
            "n": 1,
            "name": "spectral_flux"
        }
    }
}
```

Listing 5.7: Attribute Metadata for Spectral Flux

5.3 Interface Design Challenges

Many current API and interface decisions are motivated by attempts to replicate functionality seen in other research papers. This section will give a discussion of some of the challenges we faced, and the reasoning behind the design decisions we made for the interface. For example, the decision to incorporate a metadata file was conceived as a solution to several of the problems listed below.

5.3.1 Attribute Extractor Functionality

The template specification for attribute extraction has evolved along with the breadth of functionality supported. The first iterations for the attribute extractor had a very simple template:

```
def extract_attribute(data):
    ...
    return attribute
```

Listing 5.8: Attribute Extractor template in early iterations

where

- `data` is 2D array
- `attribute` is a single numeric value

Note that, in this version, the data is expected to be a single 2D array, and the function is expected to return a single numeric value.

Supporting attribute extraction on consecutive segments

Many applications involve attributes which require multiple consecutive segments to compute in order to capture dynamic changes across time. For example, audio signal analysis involves a feature called spectral flux, which is computed by taking the Euclidean distance between two consecutive frames or segments.

Clearly, Listing 5.8 would not support functions like these. We would first need to modify the input specification for `data` to support a tuple of 2D arrays, but simply changing the specification doesn't guarantee that the functionality is supported. There is no way for SenseML to figure out how many consecutive segments to process and input into the function at one time solely by looking at the attribute function. Therefore, we must require users to explicitly state the number of consecutive segments that a function requires, along with the position of the starting segment (left, middle, or right). To enable this, we introduced the metadata file. When users create a new function, they must specify, as metadata, how many consecutive segments the function is expecting. SenseML can then read this file and provide the corresponding amount of segments in the function.

Multiple Outputs

Listing 5.8 only supports functions that return a single value. However, it is sometimes beneficial to extract multiple attributes in a single function. This may be the case if a list of attributes all share a computationally intensive code base, or if certain attributes depend on the computation of others. An example of this can be seen when computing spectral centroid and spread. Computing the spectral spread of a frame requires the prior computation of the spectral centroid. Thus, in order to reduce the amount of redundant computation, it seems ideal to allow an attribute extractor function to return both of these values at the same time. Additionally, other attributes exist where the number of outputs depends on a hyperparameter. One example is extracting n Mel-frequency cepstral coefficients (MFCC).

The main difficulty in allowing attribute extractors to return multiple outputs is resolving how to name these resulting attributes in a meaningful way. Previously, SenseML only allowed users to provide a name that would correspond to a single attribute function. Now, with the metadata file, we can support more sophisticated ways to name attributes. Users can specify the output names for a function in the metadata file using the following simple guidelines:

- For a function with a single output, users must provide a single output name.
- For a function with a fixed number of outputs (greater than 1), users must provide a list of output names.
- For a function where the number of outputs depends on a hyperparameter, users must provide a name to act as a prefix. SenseML will automatically generate the output names by appending `"_i"` count to the prefix name.

Dealing with Multi-Channel Datafiles

Signal datafiles can often have multiple channels of data – for example, most sound recordings tend to come as stereo audio files. Thus, SenseML must allow for the processing of multi-channel data files.

One of the main challenges of supporting multi-channel data file processing is figuring out how to robustly ensure that the attribute extraction functions selected by a user is operable on the input data provided. SenseML's flexible design allows different attribute extraction functions to operate on independent assumptions about the input data. Some attributes may only be extracted on monochannel signals, while others may be computed using sophisticated methods on multiple channels. Depending on the implementation, if there is a mismatch between the number of channels a function expects and the number actually provided in the input data, the attribute will fail to extract and the function will throw an error.

To address this, SenseML requires users to indicate the number of channels that a function expects as an input in the metadata file. Before running each of the attribute extraction functions, SenseML will read this metadata to validate that each of the attributes that the user provides expects the same number of channels as the input data.

5.3.2 Usability Considerations

User function validation

It is important that SenseML validate the functions and inputs provided by users, so that the system operates as intended. Because these functions are being run on remote EC2 hosts, if they are syntactically or operationally incorrect, the program running on the remote hosts will crash without the user's knowledge. To remedy this, `SenseMLClient` validates all user-provided functions on the user's local machine before launching remote EC2 instances. Additionally, Sense-ML provides an API that allows users to test their functions and inputs on sample datafiles before committing to the cloud system run.

```
S.validate_function(func, func_type, hyperparams, data_file)
```

Usability in creating functions

Another usability concern is whether and how to allow people to import random Python modules when creating their functions. If users import a module or use a software that the default AMI does not currently support, then the EC2 worker instances would not be able to run the function. As of now, there is no way to automate the incorporation of new libraries, as AWS does not provide an API to do so. Thus, users will be required to update their AMI manually. This can be done by following these rough steps:

- Launch an EC2 instance with the default AMI provided.
- Install any libraries or software required.
- Verify that the library is installed and runs properly.
- Use the Amazon EC2 console to create a new AMI from that EC2 instance.

Users must then use this updated AMI id as a config to the `SenseMLCLient`.

Chapter 6

Urban Sound Classification Dataset

To demonstrate Sense-ML, we decided to use a real world dataset and replicate data preprocessing steps from [17]. We intend to release the resulting attribute time series database for public use.

6.1 Urban Sounds Dataset

We are working with the *UrbanSound* dataset, which contains 27 hours of audio with 18.5 hours of manually labelled sound occurrences over 10 sound classes: air conditioner, car horn, children playing, dog barking, drilling, engine idling, gunshot, jackhammer, siren, and street music.

We also tested the system on the *UrbanSound 8k* repository, which contains a more tailored and processed dataset based on research done in [9]. In this research it was determined that 4 seconds is a sufficient amount of time for subjects to identify environmental sounds. To reflect this, each sound file in the *UrbanSound8k* dataset consists of at most a 4-second data slice from a corresponding file in the *UrbanSound* dataset.

6.2 Preparing the S3 Bucket

In order to run SenseML, we must make sure that the data is stored in the correct format on our S3 bucket. Each signal file must be of the format

```
s3://<s3-bucket>/data/<entity-id>/<file-id>
```

For the Urban Sounds dataset, the sound files are provided as

```
audio/<class-type>/<file-id>.wav
```

There is no notion of an entity, which is a required signal descriptor for populating the database tables. Therefore, we decide to treat each individual signal as its own entity, and use the filename (i.e. `<file-id>`) as the entity ID by mapping each file from `audio/<class-type>/<file-id>.wav` to `data/<file-id>/<file-id>.wav`

For the Urban Sounds 8k dataset, the sound files are named according to the following outline:

```
fold*/<file-id>.wav
```

where `file-id` = `<entity-id>-xx-xx-xx`. We map each file as named above to `data/<entity-id>/<file-id>.wav`

After the file name remapping is completed for each sound file and the function repository is uploaded as mentioned in the previous chapter, our S3 bucket fulfills the requirements to run the Sense-ML stages.

6.3 Running SenseML

6.3.1 The Segmenter Function

One standard procedure for segmenting an audio signal in order to perform sound classification is to use a fixed analysis frame. ([17], [10], [8]). This can be formalized as a sliding window with a fixed percent overlap between frames (or segments). The length of the window and the amount of overlap between frames are both chosen depending on the relevant application.

For our segmenting function, we decided to keep these two variables as hyper-parameters, so that users would be able to modify them according to their desired application. Listing 6.1 shows the implementation of the sliding window segmentation algorithm, using scipy.io.wavfile.read() as the signal decoding function.

```python
import scipy.io.wavfile as wav
import os

def segment(f, window_size, percent_overlap):
    """
    params:
        window_size = size of window in milliseconds, float
        percent_overlap = amount of frame overlap for segments,
   float
    """

    samp_freq, data = wav.read(f)
    num_channels = data.ndim
    num_samples = data.shape[0]
    start_intervals = []
    segments = []

    window = int(samp_freq * window_size/1000.0);
    offset = int((1-percent_overlap) * window)
    num_segments = num_samples // offset

    for i in range(num_segments):
        x = i * offset
        start_intervals.append(x)
        segment = data[x:x+window]
        segments.append(segment.reshape(len(segment), num_channels))
    return samp_freq, start_intervals, segments, num_channels
```

Listing 6.1: audio.py segmenter function

6.3.2 Attribute Extraction Functions

The field of audio classification has been studied for decades. The current techniques involve extracting a variety of short term features such as energy, ZCR (Zero Crossing Rate), spectral features, statistical moments, and MFCC's. ([18], [10], [8]). We were able to find an implementation of a majority of these features in a python library called pyAudioAnalysis ([20]). As the library is open-sourced under Apache License, we adopted the functions into our system while giving credit to the author, tyiannak.

We selected a subset of these features that would be representative of the different types of attribute functions that SenseML supports. The following is a list of attributes we decided to extract for our test run:

- Energy,
- Entropy,
- Spectral Entropy,
- Spectral Roll-off,
- Spectral Flux,
- Spectral Centroid and Spread,
- Zero Crossing Rate,
- MFCCs,

For each of these attributes, we had to create a new extractor function and add an entry in the metadata file. We will walk through the process of creating and integrating the functions into SenseML for a few of these to showcase the variety of attribute functions that SenseML supports.

Spectral Entropy

Listing 6.2 shows the code for extracting spectral entropy.

```python
import numpy as np
from scipy.fftpack import fft

eps = 0.00000001
... # code snippet truncated
...

def extract_attribute(data, numOfShortBlocks):
    """Computes the spectral entropy"""
    data = np.array(data[0])
    data = np.mean(data, axis=1)

    nFFT = len(data)/2
    X = FFT(data, nFFT)
    L = len(X)
    Eol = np.sum(X ** 2)

    subWinLength = int(np.floor(L / numOfShortBlocks))
    if L != subWinLength * numOfShortBlocks:
        X = X[0:subWinLength * numOfShortBlocks]

    subWindows = X.reshape(subWinLength, numOfShortBlocks, order='F'
).copy()
    s = np.sum(subWindows ** 2, axis=0) / (Eol + eps)
    En = -np.sum(s*np.log2(s + eps))

    return En
```

Listing 6.2: Spectral Entropy Attribute Extractor Function

The main points to note about the spectral entropy function are

- it has a required hyperparameter: `numOfShortBlocks`
- the output of the function is a single variable

These points translate the following respective metadata:

- "hyperparams": ["numOfShortBlocks"]
- "output": {"type":"fixed", "n": 1, "name":"spectral_entropy"}

Listing 6.3 is the resulting JSON object appended to the metadata file.

```json
{
    "spectral_entropy.py": {
        "domain": "audio",
        "description": "Spectral Energy",
        "hyperparams": ["numOfShortBlocks"],
        "nChannels": 2,
        "nConsecutive": {
            "n": 1,
            "position": "middle"
        },
        "includeFS": false,
        "output": {
            "type": "fixed",
            "n": 1,
            "name": "spectral_entropy"
        }
    }
```

Listing 6.3: Metadata for spectral_entropy.py

Spectral Centroid and Spread

The spectral centroid and spread function is an example where users may want to extract multiple attributes with a single function. As seen in Listing A.1, the computation of the spectral spread requires that the spectral centroid has also been computed. Thus separating these two functions would result in a lot of redundant spectral centroid computations.

In the metadata file, we need to specify that the function will return two attributes and provide the names for these functions. We can do this by setting the output metadata as follows

```
"output": {
    "type": "fixed",
    "n": 2,
    "name": ["spectral_centroid","spectral_spread"]
}
```

Another note is that the function is expecting SenseML to provide the sampling frequency as an additional parameter. This is encoded in the metadata by setting the `includeFS` parameter to `true`. Listing A.4 is the resulting metadata for the spectral centroid and spread extracter.

Mel-frequency cepstral coefficients

The MFCC function is example of an attribute function where the number of outputs is determined by a hyperparameter. To do this, we must indicate the output type and naming prefix in the metadata.

```
"output": {
    "type": "variable",
    "namePrefix": "mfcc"
}
```

See Listing A.3 for the full code used for extracting MFCCs and Listing A.4 for the corresponding metadata for the function.

6.3.3 Running Populator w/ Attribute Extraction

Now that we have defined the segmenters and attribute extractors, we can proceed with running the SenseMLClient. We used 20 EC2 worker instances of type r4.large (that was the maximum number of instances our account was allowed for). Listing 6.4 shows the populate function as used during the run.

```
S.populate(segmenter = ("audio.py", {"window_size": 23.2, "
   percent_overlap": 0.50}),
            sample_frequency = 44100,
            attributes = [
                ('zcr.py',  {}),
                ('energy.py', {}),
                ('energy_entropy.py',{"numOfShortBlocks":10}),
                ('spectral_rolloff.py', {"coeff":0.9}),
                ('spectral_flux.py', {}),
                ('spectral_centroid_and_spread.py', {}),
                ('spectral_entropy.py', {"numOfShortBlocks":10}),
                ('mfcc.py', {"nceps":20})
            ],
           init_tables = True)
```

Listing 6.4: Running Populator with selected segmenter and attributes

Note that we are running the populate module with the optional attribute parameter to do the attribute extraction at the same time instead of running the two stages separately. Doing this is always faster than running the two stages separately.

6.4 Performance Evaluation

We ran two experiments, one for the `UrbanSound` dataset, and the other for the `UrbanSound8k` dataset. For each signal file, we extracted the 19 attributes (3 normal, 4 spectral features, and 14 MFCCs) listed in 6.3.2.

Running SenseML on the `UrbanSound8k` datasets took approximately 30 minutes to complete. This processed a total of 6.6 GB of raw signal data and generated total of 2,477,291 segments from 8774 signals.

For the `UrbanSound` dataset, SenseML ran for approximately 2.2 hours, processing 20.5 GB of raw signal files and producing 6,040,163 segments from 9655 signals.

6.5 Challenges Faced

While running SenseML on the large repository of data, there were a few challenges and limitations in the interface that had to be addressed.

6.5.1 Different types of input files

When working with a vast repository of data, it is very likely that the data files will not all be in the same file format. Although the *UrbanSound8k* repository only consisted of wav files, the original *UrbanSound* repository had a combination of mp3 and wav files. This becomes problematic when segmenting the data files, as typically, segmenter files are not tailored to read a majority of data file types. For the segmenter function in Listing 6.1, when reading the raw signal file into a data array, we used the scipy.io.wavfile python module, which is not able to parse mp3 data files. Thus, during our run, we decided to only operate on wav files.

Additionally, even within the same data file format (e.g. wav), there were no guarantees of a successful data file decoding. There were certain wav files that were not readable with the scipy wavfile library. In particular, wav samples encoded as 4-bit MS ADPCM appear to require specific software to decode.

One solution is to have a more rigorous signal decoding component in each segmenter. However, this would require that you know the different data types for all the data files you are working with, which may or may not be a reasonable assumption. Another similar potential solution is to change the signature of the segmenters to take in a data array as input instead of a file, thereby separating the signal decoding and the segmenting process. This would mean that the segmenter is agnostic to the filetype beforehand and will only operate on raw data samples.

The initial reasoning for having the file be the input to the segmenter was to give users more flexibility in implementing the segmenting process. Having a file as input would allow people to use third party software to run specific segmenting algorithms that operate on raw data files.

Chapter 7

Conclusion

SenseML provides a flexible and scalable solution to the problems of large scale IoT sensor data preprocessing. SenseML maintains a repository of commonly used signal preprocessing functions, allowing users to clean, segment, and extract attributes to achieve a compressed, low-frequency attribute time-series across multiple domains and applications. In this thesis we achieved the following objectives:

- Developed an end-to-end solution for the IOT signal processing.
- Established a framework to enable contributions from domain experts.
- Developed a SenseML client and a set of `apis` for different steps in a signal processing pipeline.
- Demonstrated the platform on the UrbanSound dataset.
- Created the open source repository and established the infrastructure for software sustainability.

7.1 Future Work

There is still a lot of work that can be done to improve Sense-ML. The eventual goal is for a robust, easy-to-use platform for analyzing IoT signals.

- Improve the contribution framework and testing experience.
- Identify interesting IOT datasets and create attribute time series databases.
- Enhance the repository of available functions.
- Provide `api`s to incrementally build the database.
- Provide `api`s to tune hyperparameters for the attribute and segmenter functions.
- Develop the machine learning functionality that works on top of the the attribute time series in DynamoDB.

Appendix A

Code Listing

```python
import numpy as np
from scipy.fftpack import fft

eps = 0.00000001
def FFT(data, nFFT):
    X = abs(fft(data))                                   # get fft
 magnitude
    X = X[0:nFFT]                                      # normalize fft
    return X / len(X)

def extract_attribute(data, fs):
    """Computes spectral centroid of frame (given abs(FFT))"""
    data = np.array(data[0])
    data = np.mean(data, axis=1)

    nFFT = len(data)/2
    X = FFT(data, nFFT)

    ind = (np.arange(1, len(X) + 1)) * (fs/(2.0 * len(X)))

    Xt = X.copy()
    Xt = Xt / Xt.max()
    NUM = np.sum(ind * Xt)
    DEN = np.sum(Xt) + eps

    # Centroid:
    C = (NUM / DEN)

    # Spread:
    S = np.sqrt(np.sum(((ind - C) ** 2) * Xt) / DEN)

    # Normalize:
    C = C / (fs / 2.0)
    S = S / (fs / 2.0)

    return (C, S)
```

Listing A.1: Spectral Centroid and Spread Attribute Extractor Function

```

{
    "spectral_centroid_and_spread.py": {
        "domain": "audio",
        "description": "Spectral Centroid of Frame",
        "hyperparams": [],
        "nChannels": 2,
        "nConsecutive": {
            "n": 1,
            "position": "middle"
        },
        "includeFS": true,
        "output": {
            "type": "fixed",
            "n": 2,
            "name": ["spectral_centroid","spectral_spread"]
        }
    }
```

Listing A.2: Metadata for mfcc.py

```
import numpy as np
from scipy.fftpack.realtransforms import dct
from scipy.fftpack import fft

eps = 0.00000001
def FFT(data, nFFT):
    X = abs(fft(data))                                    # get fft
  magnitude
    X = X[0:nFFT]                                      # normalize fft
    return X / len(X)

def extract_attribute(data, fs, nceps):
    data = np.array(data[0])
    data = np.mean(data, axis=1)
    nFFT = len(data)/2
    X = FFT(data, nFFT)

    fbank, freqs = mfccInitFilterBanks(fs, nFFT)

    ceps = MFCC(X, fbank, nceps)
    return ceps

def mfccInitFilterBanks(fs, nfft):
    """
    Computes the triangular filterbank for MFCC computation (used in
     the stFeatureExtraction function before the stMFCC function call
  )
    This function is taken from the scikits.talkbox library (MIT
  Licence):
    https://pypi.python.org/pypi/scikits.talkbox
    """

    # filter bank params:
    lowfreq = 133.33
    linsc = 200/3.
    logsc = 1.0711703
    numLinFiltTotal = 13
    numLogFilt = 27

    if fs < 8000:
        nlogfil = 5

    # Total number of filters
    nFiltTotal = numLinFiltTotal + numLogFilt

    # Compute frequency points of the triangle:
    freqs = np.zeros(nFiltTotal+2)
    freqs[:numLinFiltTotal] = lowfreq + np.arange(numLinFiltTotal) * linsc
    freqs[numLinFiltTotal:] = freqs[numLinFiltTotal-1] * logsc ** np.arange(1, numLogFilt + 3)
    heights = 2./(freqs[2:] - freqs[0:-2])

    # Compute filterbank coeff (in fft domain, in bins)
    fbank = np.zeros((nFiltTotal, nfft))
    nfreqs = np.arange(nfft) / (1. * nfft) * fs

    for i in range(nFiltTotal):
        lowTrFreq = freqs[i]
        cenTrFreq = freqs[i+1]
        highTrFreq = freqs[i+2]

        lid = np.arange(np.floor(lowTrFreq * nfft / fs) + 1, np.floor(cenTrFreq * nfft / fs) + 1, dtype=np.int)
        lslope = heights[i] / (cenTrFreq - lowTrFreq)
        rid = np.arange(np.floor(cenTrFreq * nfft / fs) + 1, np.floor(highTrFreq * nfft / fs) + 1, dtype=np.int)
        rslope = heights[i] / (highTrFreq - cenTrFreq)
        fbank[i][lid] = lslope * (nfreqs[lid] - lowTrFreq)
        fbank[i][rid] = rslope * (highTrFreq - nfreqs[rid])

    return fbank, freqs


```

```python
def MFCC(X, fbank, nceps):
    """
    Computes the MFCCs of a frame, given the fft mag
    ARGUMENTS:
        X:          fft magnitude abs(FFT)
        fbank:      filter bank (see mfccInitFilterBanks)
    RETURN
        ceps:       MFCCs (13 element vector)
    Note:    MFCC calculation is, in general, taken from the scikits
   .talkbox library (MIT Licence),
    #    with a small number of modifications to make it more
   compact and suitable for the pyAudioAnalysis Lib
    """

    mspec = np.log10(np.dot(X, fbank.T)+eps)
    ceps = dct(mspec, type=2, norm='ortho', axis=-1)[:nceps]
    return ceps
```

Listing A.3: MFCC Attribute Extractor Function

```json
{
    "mfcc.py": {
        "domain": "audio",
        "description": "MFCC",
        "hyperparams": ["nceps"],
        "nChannels": 2,
        "nConsecutive": {
            "n": 1,
            "position": "middle"
        },
        "includeFS": true,
        "output": {
            "type": "variable",
            "namePrefix": "mfcc"
        }
    }
}
```

Listing A.4: Metadata for mfcc.py

Bibliography

[1] Amazon web services. https://aws.amazon.com/. Accessed: 2017-08-14.

[2] Aws command line interface. http://docs.aws.amazon.com/cli/latest/userguide/cli-chap-welcome.html. Accessed: 2017-08-14.

[3] Event prediction, anomaly detection, and internet of things (iot). http://simularity.com/solutions/industries/. Accessed: 2017-08-14.

[4] pyaudioanalysis. http://docs.aws.amazon.com/cli/latest/userguide/cli-chap-welcome.html. Accessed: 2017-08-14.

[5] Sox - sound exchange. http://sox.sourceforge.net/. Accessed: 2017-08-14.

[6] The wfdb software package. https://physionet.org/physiotools/wfdb.shtml. Accessed: 2017-08-14.

[7] S. Adavanne, G. Parascandolo, P. Pertilä, T. Heittola, and T. Virtanen. Sound Event Detection in Multichannel Audio Using Spatial and Harmonic Features. *ArXiv e-prints*, June 2017.

[8] Jeroen Breebaart and Martin McKinney. Features for audio classification. 2, 01 2004.

[9] S. Chu, S. Narayanan, and C. C. J. Kuo. Environmental sound recognition with time x2013;frequency audio features. *IEEE Transactions on Audio, Speech, and Language Processing*, 17(6):1142–1158, Aug 2009.

[10] David Gerhard. Audio signal classification: History and current techniques. 2003.

[11] V́ineet Gopal. Physiominer : a scalable cloud based framework for physiological waveform mining. Master's project, Massachusetts Institute of Technology, EECS Department, June-August 2014. This is a full MASTERSTHESIS entry.

[12] Laszlo Hejjel and Elizabeth Roth. What is the adequate sampling interval of the ecg signal for heart rate variability analysis in the time domain? *Physiological Measurement*, 25(6):1405, 2004.

[13] Biing-Hwang Juang. On the use of bandpass liftering in speech recognition. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 1987.

[14] Bob Kemp and Jesus Olivan. European data format 'plus' (edf+), an edf alike standard format for the exchange of physiological data. *Clinical Neurophysiology*, 114(9):1755 – 1761, 2003.

[15] Dongsheng Li, Siu-Chun M Ho, Gangbing Song, Liang Ren, and Hongnan Li. A review of damage detection methods for wind turbine blades. *Smart Materials and Structures*, 24(3):033001, 2015.

[16] Marcello R. Napolitano, Ching I. Chen, and Steve Naylor. Aircraft failure detection and identification using neural networks. *Journal of Guidance, Control, and Dynamics*, 16(6):999–1009, Nov 1993.

[17] Justin Salamon, Christopher Jacoby, and Juan Pablo Bello. A dataset and taxonomy for urban sound research. In *Proceedings of the 22Nd ACM International Conference on Multimedia*, MM '14, pages 1041–1044, New York, NY, USA, 2014. ACM.

[18] S. Sivasankaran and K. M. M. Prabhu. Robust features for environmental sound classification. In *2013 IEEE International Conference on Electronics, Computing and Communication Technologies*, pages 1–6, Jan 2013.

[19] J. X. Sun, A. T. Reisner, and R. G. Mark. A signal abnormality index for arterial blood pressure waveforms. In *2006 Computers in Cardiology*, pages 13–16, Sept 2006.

[20] tyiannak. pyaudioanalysis. https://github.com/tyiannak/pyAudioAnalysis, 2017.

[21] M.H. Vafaie, M. Ataei, and H.R. Koofigar. Heart diseases prediction based on ecg signalsâĂŹ classification using a genetic-fuzzy system and dynamical model of ecg signals. *Biomedical Signal Processing and Control*, 14:291 – 296, 2014.