# PhysioMiner: A Scalable Cloud Based Framework for Physiological Waveform Mining

by

Vineet Gopal

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
June 2014

Author . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Department of Electrical Engineering
and Computer Science
May 23, 2014

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

Certified by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Una-May O'Reilly
Principal Research Scientist
Thesis Supervisor

Accepted by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Prof. Albert R. Meyer
Chairman, Masters of Engineering Thesis Committee

# PhysioMiner: A Scalable Cloud Based Framework for Physiological Waveform Mining

by

Vineet Gopal

Submitted to the Department of Electrical Engineering
and Computer Science
on May 23, 2014, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

## Abstract

This work presents PhysioMiner, a large scale machine learning and analytics framework for physiological waveform mining. It is a scalable and flexible solution for researchers and practitioners to build predictive models from physiological time series data. It allows users to specify arbitrary features and conditions to train the model, computing everything in parallel in the cloud.

PhysioMiner is tested on a large dataset of electrocardiography (ECG) from 6000 patients in the MIMIC database. Signals are cleaned and processed, and features are extracted per period. A total of 1.2 billion heart beats were processed and 26 billion features were extracted resulting in half a terabyte database. These features were aggregated for windows corresponding to patient events. These aggregated features were fed into DELPHI, a multi algorithm multi parameter cloud based system to build a predictive model. An area under the curve of 0.693 was achieved for an acute hypotensive event prediction from the ECG waveform alone. The results demonstrate the scalability and flexibility of PhysioMiner on real world data. PhysioMiner will be an important tool for researchers to spend less time building systems, and more time building predictive models.

Thesis Supervisor: Kalyan Veeramachaneni
Title: Research Scientist

Thesis Supervisor: Una-May O'Reilly
Title: Principal Research Scientist

# Acknowledgments

I would like to give a big thanks to Kalyan Veeramachaneni for his support, guidance, and feedback throughout the thesis process. His willingness to try new things and his keen eye are a large this reason this project was successful. I would also like to thank Will Drevo, Colin Taylor, and Brian Bell for being awesome lab-mates.

Outside the lab, I would like to say thank you to Annie Tang for providing support (and nourishment) when needed. Thank you to all of my friends who made my time here so special. And thank you to my parents for doing everything possible to get me where I am today.

# Contents

# Chapter 1

# Introduction

Over the past ten years, the amount of data that is collected, stored, and analyzed on a daily basis has grown exponentially. Ninety percent of the data in the world has been created in just the last two years [20]. Companies like Google and Facebook have petabytes of information stored in various warehouses. As storage has become extremely cheap, companies and researchers generate and store tremendous amounts of user interaction data from their products. The goal is to process and analyze this data for patterns and correlations, and build predictive models. In some cases, the amount of data has allowed researchers to build much more accurate models of real world events – everything ranging from human behavior to weather patterns.

Healthcare data has seen similar growth, as hospitals can now store terabytes of information about their patients at a reasonable price. But health care data is not limited to hospitals alone. Healthcare data comes from a variety of sources in a variety of formats. EMR (Electronic Medical Records) and EHR (Electronic Health Records) service providers like *athenahealth*, *eClinicalWorks*, and *Epic* maintain patient records including demographics, medical history, and laboratory results from clinical visits. These companies store records for millions of patients, and their mission is to be the information backbone for a patient's entire medical history [4].

A large subset of the patient data recorded by various players in the medical space is *physiological* signals. ICU's monitor various physiological signals continuously throughout the patient's stay, in a hope to better medical diagnosis, enhance future care, or personalize the care space. Measuring physiological signals like Arterial Blood Pressure (ABP) and Electroencephalography (EEG) are invasive procedures, requiring electrodes placed on the scalp or the insertion of catheters into an artery. Other physiological signals like Electrocardiography (ECG)) are less invasive, and simply require placing on a monitor on the heart or fingertips of the patient. Because ECG is less invasive, it is becoming increasingly popular in collecting data outside of the hospital. Fitbit, for example, allows users to put on a wristband to record and store personal health data throughout the day [11]. Alivecor created a specialized heart monitor (that acts as an iPhone cover) that is both affordable and easy to use in various settings [1]. While Alivecor is prescription only, Cardiac Design is an Over-The-Counter iPhone heart monitor that users can use to record their own ECG signals [5]. Qardio lets users record both BP and ECG data using two specialized products, a wireless blood pressure cuff and a chest strap-on device [14]. With the myriad of companies starting to get into the physiological analysis space, petabytes of data are being collected everywhere from the home, to hospitals, to the gym.

Given the large amount of data from multiple health data vendors, the machine learning community has been particularly interested in health care data. A subset of researchers and practitioners are interested in physiological signals, because of their implications on society. Lets take a look at a specific example: *heart disease*. 600,000 people die from heart disease every year in the United States [9]. Some occupations are much more susceptible than others - fireman are 300 times more likely to suffer a heart attack than the average person. With the decreasing costs of monitoring technology, we can actually monitor the ECG signals of each fireman in real time. If we could predict when a fireman was likely to have a heart attack with a minutes notice, we could likely save the lives of thousands of firemen every year.

At the same time, a large amount of research has gone into predicting Acute Hypotensive Episdoes (AHE) using ABP data. An Acute Hypotensive event occurs when a

Figure 1-1: Fitbit (top left) built a wristband that lets users record personal data throughout the day. Alivecor (top right) built an iPhone app and case that allows users to record their ECG signals. Cardiac Design (bottom left) has a similar product to Alivecor, except it is over the counter. Qardio uses a blood pressure cuff to measure ABP signals.

patient's blood pressure goes below a specific threshold for a significant period of time. Physionet and Computers in Cardiology hosted a machine learning challenge for developers to predict AHE events 30 minutes into the future, given a large training data set. The resulting models were promising - accuracies of over 90% were reported.

Only a small surface of the vast amounts of healthcare data has been tapped. Ten years ago, the problems were in accumulating enough data to analyze, and finding scalable, accessible storage for the data. Now, the problems lie in how to manage and exploit the sheer scale of available data. Along with big data has come cheap storage and cloud computing. Cheap storage has allowed us to maintain high resolution data from millions of patients over extended periods of time. Data centers have allowed this storage to be scalable and anywhere accessible. Cloud computing has made it possible to build scalable applications to analyze this extremely large, and quickly growing dataset. With a few clicks of a mouse, it is just as to easy to spin up ten machines or ten thousand. Scalable cloud computing has made it possible to analyze petabytes of data in mere hours.

While the benefits of cloud computing are easily seen with building services and distributed storage systems, figuring out how to divide a data processing or machine learning pipeline into separate, parallelizable processes can take a significant amount of time and resources. Building a compute infrastructure to handle this pipeline takes even more time. A lot of focus is on building scalable customized architectures for processing data, when it should be on the processing and analysis itself.

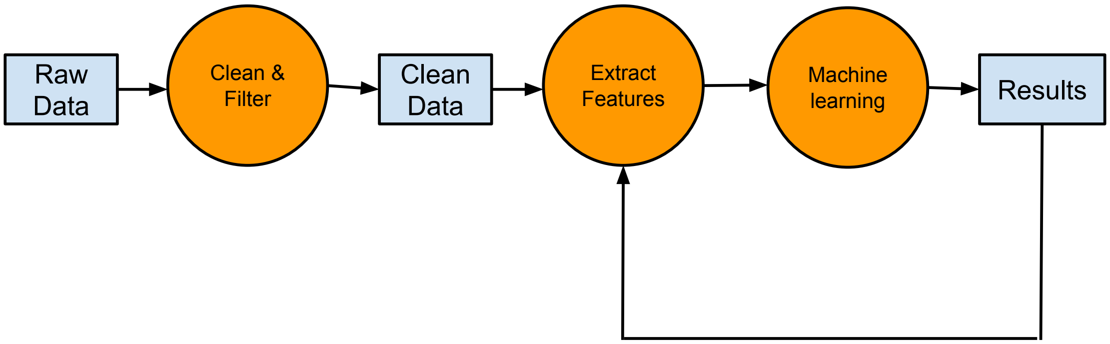

Figure 1-2: PhysioMiner acts as a black box for users. Users provide data and features, and PhysioMiner outputs a set of processed data. This processed data can be used along with analysis conditions to extract results.

We have all the tools needed to learn from this vast sea of data – a scalable cloud, cheap storage, machine learning algorithms, and the data itself. Yet, the time between data collection and analysis is often in the order of months. Researchers have to build an entire system that connects each of these tools together, often taking weeks. While starting up 1000 servers might be easy, creating a parallelized application that runs on all of them is often difficult. In the context of physiological waveforms, we observe that the analytics pipeline used by a number of researchers follow the same framework. Cleaning and filtering the data, splitting the data into periods (onset detection), extracting features (e.g. wavelet coefficients) on a per period basis, and aggregating the time series of features to perform machine learning. While the machine learning problem in focus could change, the data processing pipeline does not. Hence, a feature time series represents a good intermediate representation of data storage, which would save enormous startup times, and enable extremly scalable experimentation, discovery, and machine learning. Ideally, researchers would have an application that handled connecting the cloud and storage to data, precomputing various calculations for later use, and querying the data to investigate hypotheses. PhysioMiner is that application for physiological waveforms.

## 1.1 What is PhysioMiner?

PhysioMiner is a large scale machine learning and analytics framework for analyzing periodic time series data and creating an instance of BeatDB [18]. The periodicity of time series data allows data processing and machine learning algorithms to be easily parallelized. PhysioMiner is an application for developers to use to process and analyze their data in a scalable fashion on the cloud. This will save developers the start up costs of setting up a cloud computing framework for their time series analysis, so they can focus on the analysis itself.

PhysioMiner is split into four simple modules that allow users to:

- Populate the database with patient's physiological data
- Extract features from physiological data
- Find windows exhibiting some patient condition
- Aggregate features for different windows

PhysioMiner handles connecting the data to the cloud and database using Amazon Web Services. It abstracts away the cloud from the user, giving the user a simple command line interface to process and query data. While PhysioMiner can be used for any periodic time series data, we will focus on physiological waveforms here. A lot of research has gone into analyzing the ABP and ECG signals posted for thousands of patients in the MIMIC 2 database [12]. PhysioMiner will allow researchers to easily process and query the data at scale, while ignoring the cloud computing framework that PhysioMiner is built on. PhysioMiner will allow researchers to make the most out of the physiological data available, and will hopefully stimulate new discoveries in health care.

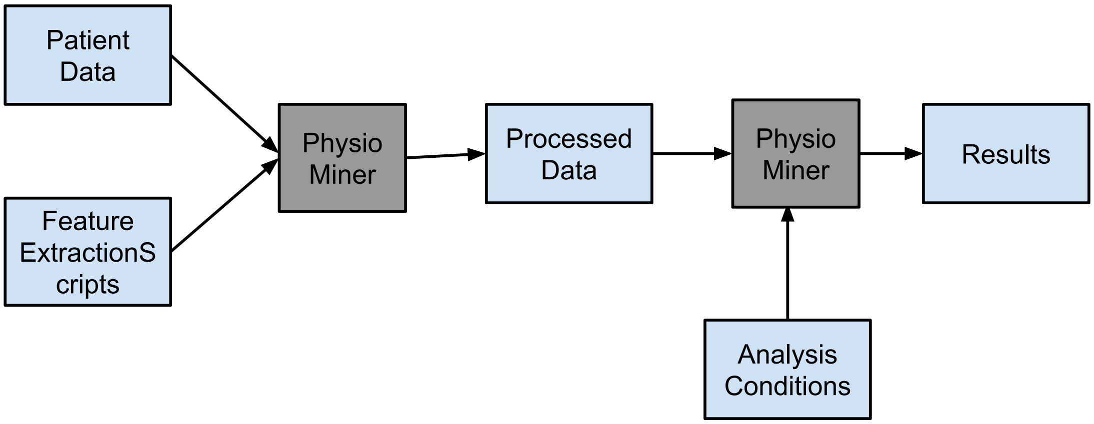

Figure 1-3: PhysioMiner acts as a black box for users. Users provide data and features, and PhysioMiner outputs a set of processed data. This processed data can be used along with analysis conditions to extract results.

## 1.2 Goals

PhysioMiner was built to be used as an analysis blackbox. There are several goals PhysioMiner must meet to achieve this.

**Usability** - It should be easily used by a variety of users - hospital data scientists, research groups, even businesses. This means it must be both simple and powerful.

**Flexibility** - Users should be able to customize their data processing and analysis. Users should be able to adjust things on the fly, without restarting the entire process. In particular, PhysioMiner should support:

- **Adding new patients** - The user should be able to add new patients into the database at any time, with all necessary data processing done automatically.
- **Adding new features** - The user should be able to extract new features from existing patients into the database at any time.
- **Adding new conditions** - The user should be able to specify new conditions to aggregate data on for analysis.

**Scalability** - It should be just as easy to process a 1MB of data as it is 1TB. In addition, analyzing 1TB of data should take roughly as long as analyzing 1MB.

**Fault tolerance** - Handling instances that fail or files with errors should be seamless to the user. The user should simply receive a list of files that were unable to be processed.

**Speed** - The processing and analysis time should never be the bottleneck for research. Time should be spent examining results, and creating new features to test, not waiting for the data to process.

**Economical** - Processing and storing terabytes of data will not be cheap, but the costs of running an analysis should be reasonable. Specifically, the expected benefits of running multiple analyses should greatly exceed the cost of running them using PhysioMiner.

# Chapter 2

# Populating the Database

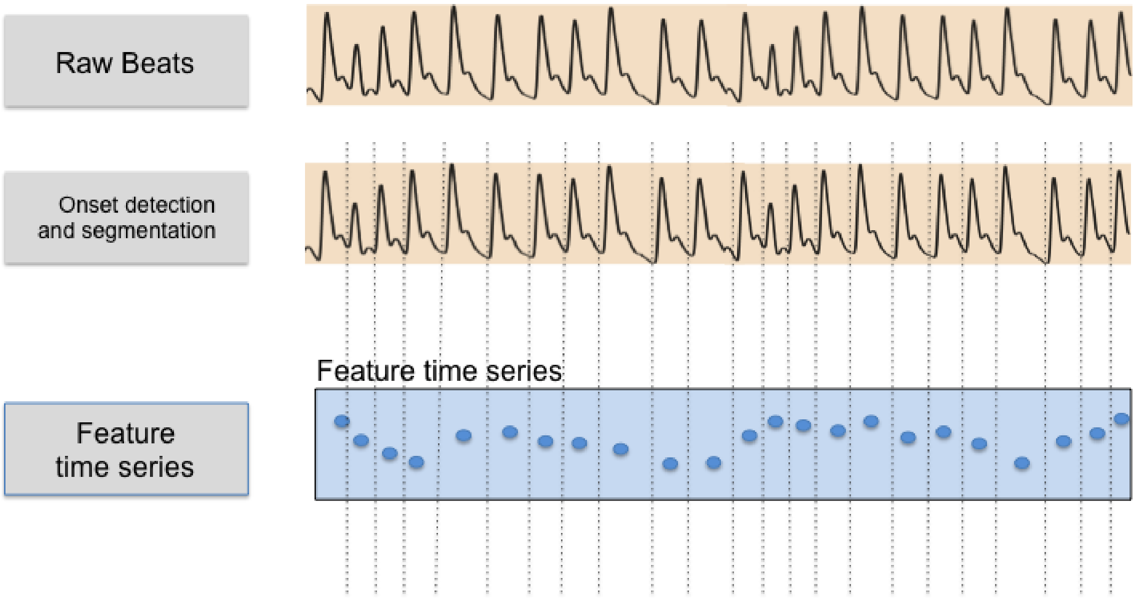

Figure 2-1: The basic pipeline for processing physiological signals. Signals come as raw waveforms. These waveforms are divided into periods, or **beats**. Features are then calculated for each beat.

The initial step in setting up a database is populating it with segmented physiological waveforms. Figure 2-1 illustrates the steps used to process and store signals. Each patient has a signal file containing readings for a particular time frame. These signals are segmented into **beats**, using an onset detector. These beats often correspond to natural periodic events like heartbeats. Features are then extracted per beat, and stored in the database.

## 2.1 System Architecture

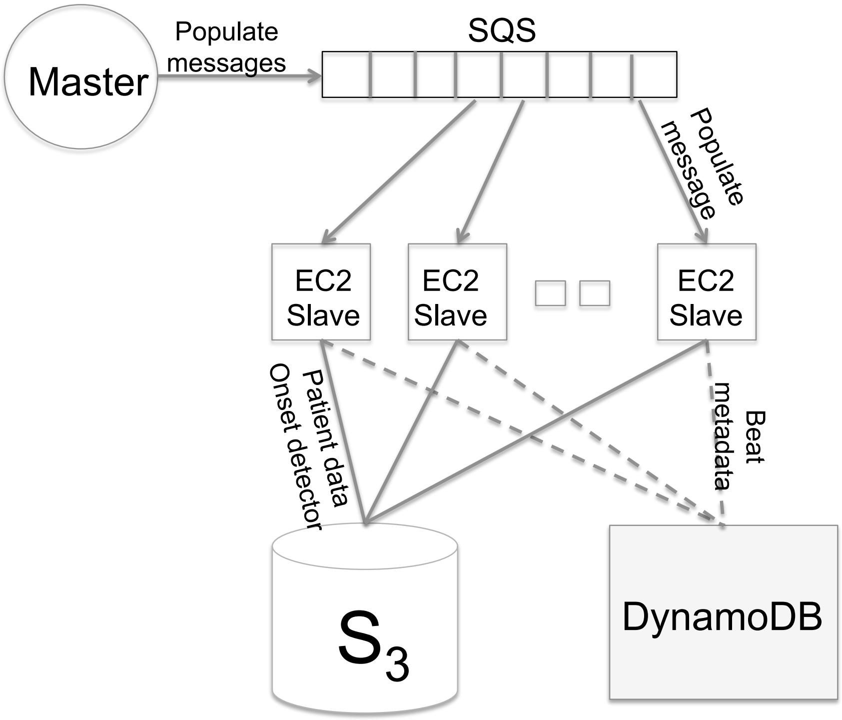

Figure 2-2: The basic system architecture for PhysioMiner. There are three main components and interactions – database, master-slave framework, and file storage.

Figure 2-2 illustrates the overall system architecture of PhysioMiner. There is a basic master-slave framework that PhysioMiner is built upon which connects to both patient file storage and the database. These components use various Amazon services to provide scalability and fault tolerance. The following is a list and brief description of each of the services that is used:

**DynamoDB** - A NoSQL database. It is designed to be fast, scalable, and supports hash keys and range keys per table. It also provides a very simple API for interacting with the database.

**Simple Queue Service (SQS)** - A simple messaging system. It provides a messaging queue that is optimal for a master-slave workflow. The master can add tasks onto the queue as messages, and slaves can pop messages off of the queue to work on them. It is designed to handle concurrency. For PhysioMiner, the cost is effectively $0 for most workflows.

**Elastic Compute Cloud (EC2)** - A scalable cloud. It provides an easy API for starting and stopping virtual machines (instances), and provides full control over each instance. It is extremely flexible, allowing cost tradeoffs for RAM, disk space, and CPU.

**Simple Storage Service** - A scalable storage mechanism. Costs depend only on the amount of data stored, and it can be easily integrated with the various other Amazon products.

The following sections discuss each of the three components and how we built PhysioMiner using them.

### 2.1.1 Database Schema

PhysioMiner stores all of its data in NoSQL tables using DynamoDB. Each patient has a unique identifier to access the data. Signals recorded for each patient may be split among multiple files, called segments. Each segment has a list of samples for a given time period - these samples are split up into periods called beats. Each beat has a start and end time, validity, and several precomputed features. The schema is split up into three separate tables in DynamoDB.

**Segment Table** This table stores a row for each segment. A segment represents a single file of patient data stored in S3. Segment ids are globally unique. The file location corresponds to the associated signal file stored in Amazon S3.

| **Patient Id** | **Segment Id** | **File Location** |
|---|---|---|
| 3000003 | AH837bC | data/mimic2wdb/3000003 |
| 3612860 | 98HF7wj8 | data/mimic2wdb/3612860 |

**Beat Table** This table stores a row for each beat. Each segment is divided into a list of beats, representing contiguous segments of signal data. For each beat, there are several columns of metadata – start sample id, end sample id, validity. There is an additional column for each feature extracted for the beat. Patient id is redundantly stored with beats, since DynamoDB does not support multiple hash keys or *JOIN* operations. This allows us to quickly identify which patient a beat belongs to, without scanning the segment table [7].

| **Patient Id** | **Segment Id** | **Beat Id** | **Interval Start** | **Interval End** | **Valid** | **Feature 1** |
|---|---|---|---|---|---|---|
| 3000003 | AH837bC | 108 | 173119 | 173208 | True | 183.32 |
| 3000003 | AH837bC | 109 | 173209 | 173292 | True | 185.94 |

**Window Table** This table stores a row for each window. Windows are defined as periods of time where a certain condition has happened to a patient (i.e. arrhythmia). There is a different window table for each condition. Each window starts and ends at specific beats on a particular segment. The features of these beats can be aggregated via various functions (mean, kurtosis, etc.) and stored as aggregated features in the table. Each window is also labeled with a classification, so a machine learning algorithm, like logistic regression, can easily be run using this table.

| **Window Id** | **Segment Id** | **Start Beat Id** | **End Beat Id** | **Class** | **Agg. Feature 1** |
|---|---|---|---|---|---|
| IFJ83G | AH837bC | 108 | 1502 | AHE | 392.1 |
| 837HR | AH837bC | 10388 | 1322 | Normal | 392.1 |

### 2.1.2 File Storage

PhysioMiner uses Amazon S3 to store patient files. Users can upload each of their patient files to a folder in S3. Several tools exist to upload files, or users can do it manually with the S3 Management Console [3]. Each file should correspond to exactly one patient, and the name of the file should be the patient id. Currently, PhysioMiner has built in support for ABP and ECG files stored in the European Data Format (EDF). Otherwise, signal files should be in a CSV format with a (sample_id, value) listed on each line. An example signal file may look like:

```
>>> cat signal.csv
1,29.8
2,30.1
3,31.2
4,34.5
10,36.7
```

Users also store feature extractor scripts and onset detectors in S3. Users specify the script location to PhysioMiner on initialization (see Section 2.2.1. These scripts are then downloaded and used by the EC2 instances (slaves) to process the signals and extract features.

### 2.1.3 Master Slave Framework

PhysioMiner is structured as a master-slave framework using Amazon SQS. Operations on the database are divided up into tasks, each of which can be encoded in a message. These message are put into a global message queue. The slaves continuously read messages from this queue, and perform the tasks associated with them. Most of these tasks involve some sort of data processing and writes to single global database. The basic workflow is shown in Figure 2-3.

#### Master

The master is responsible for creating tasks. A job is broken down into tasks, and the master populates the message queue with these tasks. PhysioMiner breaks down each job (populating, extracting, conditioning, aggregating) into parallelizable tasks. Each task is encoded as a message, and placed on the message queue for slaves to read. The master can be run on any machine (EC2 instance, laptop, etc.) Typically, the division of tasks and population of the queue takes no more than a couple of minutes.

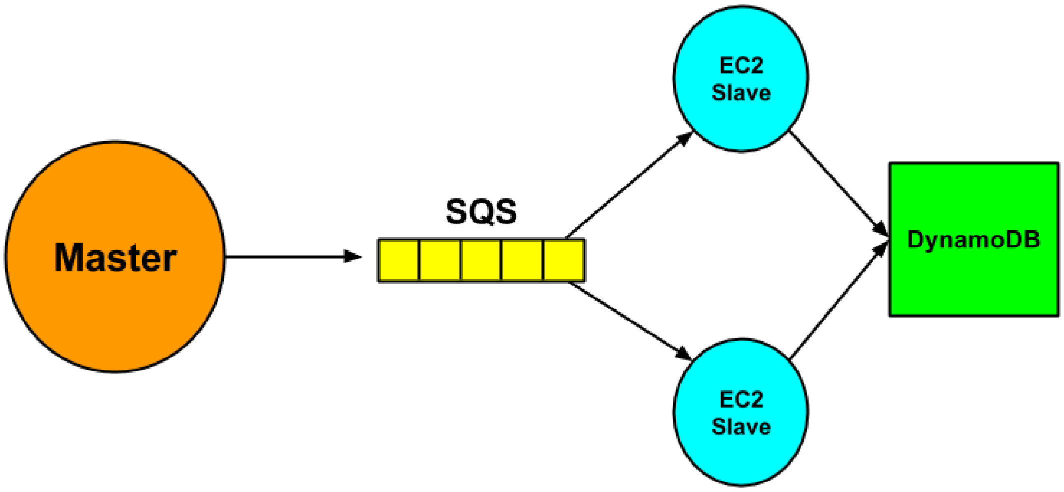

Figure 2-3: The master slave framework that PhysioMiner uses. Messages are sent from the master to the slaves through Amazon SQS, and the slaves write the output of each task to the database.

### Slave

The slave is responsible for completing the tasks in the message queue. When a slave starts running, it listens to the message queue for messages. Once it receives a message, it parses the message to extract the task to complete. The slave then finishes the task, stores the results in the database, and deletes the message from the queue.

PhysioMiner uses an Amazon EC2 instance for each slave. Each instance is its own virtual machine, with its own memory and disk storage. Users can use PhysioMiner to create these instances automatically. Users can specify the number of instances they want to use during their analysis, and the type of instance. Each instance type has different amounts of RAM, CPU power, and disk space. Table 2.1 compares the specifications and costs of each instance that was considered.

| **Instance Type** | **CPU (ECUs)** | **RAM (GiB)** | **Cost per hour** |
|---|---|---|---|
| c3.large | 7 | 3.75 | $0.105 |
| c3.xlarge | 14 | 7.5 | $0.210 |
| c3.2xlarge | 28 | 15 | $0.420 |
| r3.large | 6.5 | 15 | $0.175 |
| r3.xlarge | 3 | 0.5 | 0.350 |
| r3.2xlarge | 26 | 61 | $0.700 |

Table 2.1: The per hour costs of different Amazon EC2 instances.

Amazon EC2 Instances are created using an Amazon Machine Image (AMI). An AMI is a snapshot of an EC2 instance, and can be used to initialize new instances in the same state. We have created a public AMI for PhysioMiner that contains all of the installations necessary to run the PhysioMiner software. These include Python, Java, and various Physionet software [17]. These images also come preloaded with the software stack required by the slaves as discussed in Section 2.1.3. When PhysioMiner creates the instances, it automatically starts the slave program on the instance using the *user data* framework provided by Amazon [10].

PhysioMiner uses this public AMI as the default image for its slaves. However, users can specify an optional argument with the id of the AMI they want to use instead. This allows users to configure an instance themselves, and install any software necessary to run their code. To do this, a user can create an instance using the public PhysioMiner image, configure it as desired, and take a snapshot of the new image. This image does not need to be public, so users with privacy concerns can simply set the new AMI to be private.

### Amazon SQS

Amazon SQS provides a variety of services to make the master-slave framework easier to build. Once a message is read by a slave, the message is marked as not visible. This prevents other slaves from performing the same task. Once the slave is done with the task, it deletes the message from the queue. It also provides a fault tolerance mechanism - once a slave reads a message and marks it as not visible, it has some allotted time to perform the task and delete the message. If the slave does not complete the task in time, the message is marked as visible automatically, and other slaves can perform the task. This mechanism allows creates a robust, fault tolerant architecture - if a slave dies, or cannot successfully finish a task, the task will eventually get re-added to the queue.

## 2.2 How is it used?

Populating a PhysioMiner database can be conceptually split up into two separate modules - **Populating** and **Feature Extracting**. These conceptual modules also represent how the code was structured, to provide maximum flexibility to the user. Some additional features were added to improve the speed of PhysioMiner- these are discussed in Section 2.3.

### 2.2.1 Population Module

Users initialize their PhysioMiner instance using this module.

**User Specifications**

Users specify the following items to run the populate module:

- **S3 Bucket** - container in S3 for any data PhysioMiner needs access to
- **S3 Patient Folder** - the folder in S3 which contains all of the patient signal files. Currently, PhysioMiner supports signal files stored in EDF format for its built in physiological onset detection.
- **Signal type** - PhysioMiner has built in support for ABP and ECG signals. Other signal types can be specified, but custom onset detection algorithms must be given.
- **Onset detector** - the location of the onset detection script in S3. If the signal type has built in support (ABP or ECG), then this is not necessary.

Optionally, features can be given to the populate module to compute them at initialization time. This allows PhysioMiner to be both flexible and economical by minimizing the data processing time. This is discussed more in Section 2.3.1.

### Master

The master retrieves a list of patient files from the S3 patient folder. It then creates a `PopulateMessage` for each patient, and pushes it to Amazon SQS. A sample `PopulateMessage` might look like:

```
Patient id: 3000003
S3 Bucket: mybucket
File path: data/ecg/3000003.edf
Signal type: ECG
Onset Detector: scripts/ecg/onset_detector
```

If feature extractors were given to the master, then the `PopulateMessage` also contains a mapping from feature names to an S3 file path for the feature script. This is discussed more in Section 2.3.1.

### Slave

Once a slave receives a `PopulateMessage`, it performs the following steps:

1. Downloads the corresponding signal file from S3.
2. Reads the sample values from the file.
3. Performs onset detection to split the signal into beats.
4. Optionally, extracts features from the beats.
5. Stores the beats into DynamoDB.

### Customization

PhysioMiner was designed to be flexible and customizable, so users can specify custom onset detection scripts to run on their signals. PhysioMiner accepts arbitrary binary executables as onset detectors, which take as input a csv file of (sample id, value), and output the starting sample id for each beat on a separate line. A typical use of the program might look something like:

```
>>> ./my_program signal.csv
    5
    134
    257
    381
    ...
```

This gives the user complete control over their data. To test the onset detection in the environment it will be run in, the user can simply spin up an EC2 instance using the default PhysioMiner AMI and perform his tests there. PhysioMiner uses the PhysioMiner AMI by default for its slave EC2 instances. However, it provides a simple interface to create PhysioMiner slaves using different AMIs (see Section A.1).

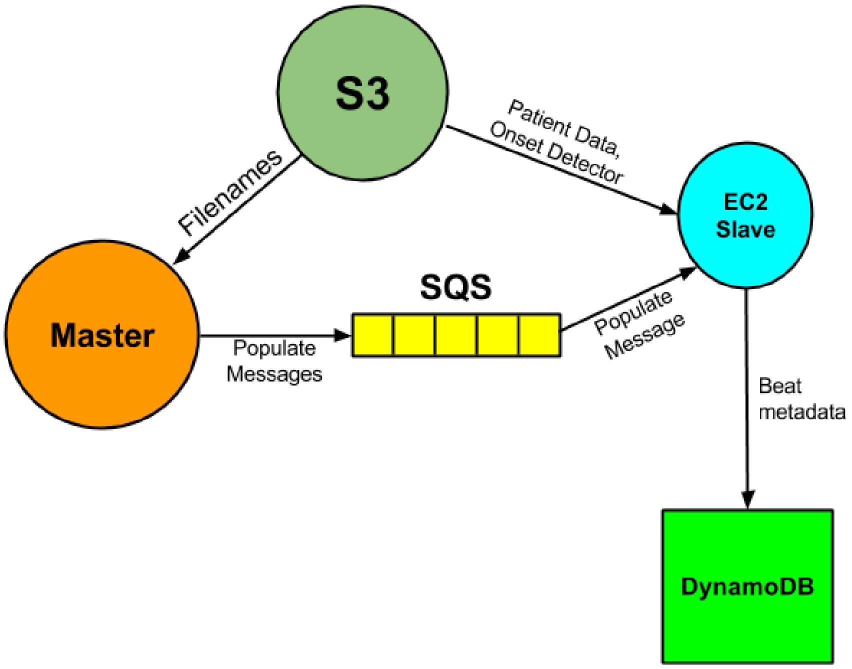

Figure 2-4: The framework that the Populate Module uses. Each message corresponds to a patient file to process. The slave downloads the file from S3, processes it, and stores the beat metadata into the database.

### 2.2.2 Feature Extract Module

Users can extract features from existing patients in the database using this module. PhysioMiner stores features per beat.

#### User Specifications

To extract features from all of the existing patients in the database, users specify the following items:

- **S3 Bucket** - container in S3 for any data PhysioMiner needs access to
- **Features** - a mapping from feature names to the script location in S3

A feature extractor is a function that takes as input a list of beat values, and outputs a single feature value. Feature scripts can be arbitrary binary executables. A typical use of a feature extractor might look like:

```
>>> ./feature_extractor 21.1 23.0 22.88 25.2 4 32.7 41.2 39.8 42.2
    5
```

#### Master

The master retrieves a list of patient ids from the database. It then creates a `FeatureExtractMessage` for each patient, and pushes it to Amazon SQS. A sample `FeatureExtractMessage` might look like:

```
Patient id: 3000003
S3 Bucket: mybucket
Features: [(mean, scripts/mean), (max, scripts/max)]
```

#### Slave

Once a slave receives a `FeatureExtractMessage`, it performs the following steps:

1. Downloads the patient file from S3.
2. Reads the sample and signal values from the file.
3. Divides the samples into beats using the beat metadata in the database.
4. Extracts features from each beat.

5. Adds the features to existing rows in the database.

### Customization

Allowing arbitrary feature extraction scripts allows the user complete control over the extracted features. It allows users to write their scripts in any language, as long as they update the slave EC2 AMIs to support it. While this provides complete flexibility, this caused significant speed challenges for PhysioMiner. Section 2.3.2 describes these challenges, and the changes made to overcome them.

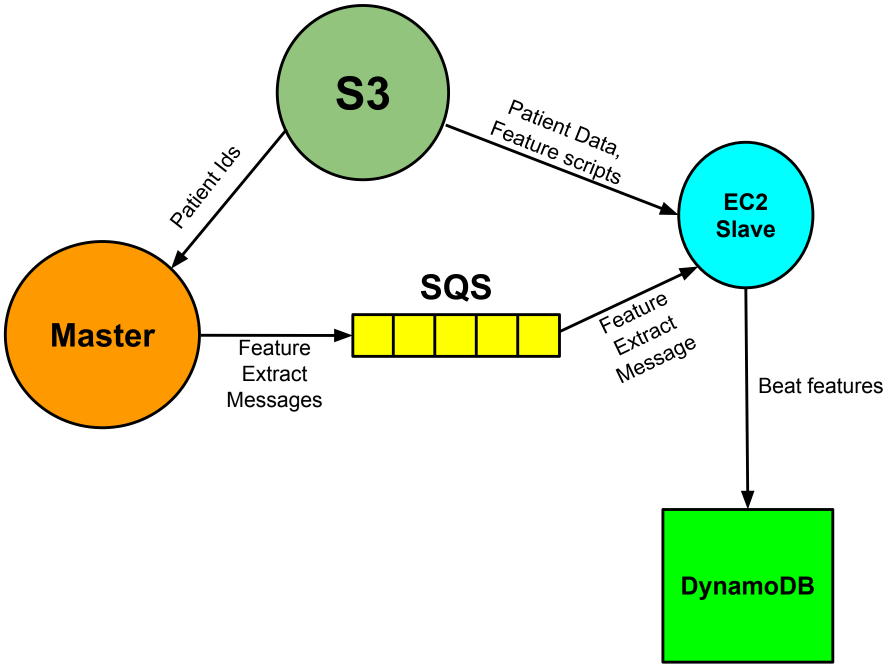

Figure 2-5: The Amazon framework that the Feature Extract uses. Each message corresponds to a segment of the database to process. For each patient in the segment, the slave will download the file from S3, process it, and store the features into the database.

## 2.3 Architectural Challenges

The challenges in implementing PhysioMiner stem directly from its goals - *speed*, *cost*, and *flexibility*. These often create conflicting axes for design. In some cases, PhysioMiner was able to accommodate both goals, while in others, tradeoffs were necessary. Other software concerns like modularity also played a significant role in the implementation decisions.

### 2.3.1 Flexibility vs. Speed: Supporting Different Use Cases

The division of PhysioMiner into modules gives the user a lot of flexibility in interacting with the data. Since each module can be run separately, the user can choose exactly what he wants to run. However, this division could mean a significant loss of speed.

For example, the populate module downloads the patient files, reads the samples, splits them into beats, and stores them in the database. The extract module downloads the patient files, reads the samples, reads the beats from the database, and stores the features back into the database. While dividing these into separate modules is good for specific use cases, it almost doubles the amount of downloading and processing necessary when feature extraction is done immediately after populating.

To solve this issue without sacrificing flexibility, the populate module can optionally take a list of features to extract. The populate module then extracts the features before populating the database. Feature extraction uses the same code as the extract module, but importantly does not create a new task. Instead, a single slave handles both the populate and feature extraction steps simultaneously. This prevents double work in downloading files and writing to the database, and ultimately makes feature extraction run about twice as fast. This solution provides the flexibility of separate modules with the speed of a single module.

### 2.3.2 Flexibility vs Speed: User Defined Procedures

PhysioMiner allows users to provide binary executables for onset detection and features. This gives users complete flexibility over their data processing, as it allows them to write feature extractors a language of their choice. This was also relatively easy to implement from an architectural standpoint. However, this had tremendous effects on speed.

Since PhysioMiner is implemented in Java, it supports binary executables using Javas ProcessBuilder. When extracting features via these binary executables, PhysioMiner simply creates a new process for each beat and reads the output of the users executable. However, starting a process has a non-trivial overhead, measured to be around 30ms on an Amazon EC2 Micro instance. Since a typical signal file can contain up to a million beats, extracting features from a single file would take a minimum of 30,000 seconds (8 hours). This goes against PhysioMiner's speed goals.

The total number of spawned processes had to be limited (ideally constant). To solve this problem, we sacrifice some user flexibility by limiting the supported languages of feature extractors to only Python. This could be extended to support other popular languages such as Java, Matlab, and C++.

To support feature extractors written in Python, PhysioMiner has a python aggregator script (written in Python) on the instance AMIs. The python aggregator script makes use of Pythons dynamic module loading. This script imports the feature extractor as a python module, and calls it for each beat in a segment. Structuring it this way means we only need to spawn one process for the python aggregator.

The feature extractor must contain a method called `extract_feature`, which takes in a list of values corresponding to a beat. This method must return the computed feature value. The python aggregator script is shown below.

```
# filename of the feature script
script = sys.argv[1]
module = imp.load_source("module", script)

# filename of file containing the data for each beat
data_file_path = sys.argv[2]
data_file = open(data_file_path)

# filename to output feature values to
out_file_path = sys.argv[3]
out_file = open(out_file_path , ’w’)

for line in data_file:
    # get the data for a single beat
    data = [float(s) for s in line.split(" ")]
    val = module.extract_feature(data)
    out_file.write(str(val) + "\n")
```

Using this structure reduced the feature extraction time from 8 hours to around 20 seconds for easily computable features on a million beats. Though the gains in speed (1500x) are tremendous, the tradeoffs in flexibility are also large, as they force the user to write extraction scripts in Python. This is an area where additional work could be done to expand the flexibility.

### 2.3.3 Flexibility vs. Simplicity: Variable-sized Tasks

Processing a single patient file and extracting features requires a significant amount of memory. For a file with $n$ beats, with an average of 100 samples per beat, this requires at least $800n$ bytes just to store the sample values. For a typical file with one million beats, this requires more than almost 750MB of memory. To process this data and extract features, it requires $2400n$ bytes of memory.

Unfortunately, there is wide variation in patient file sizes. Some are more than 10

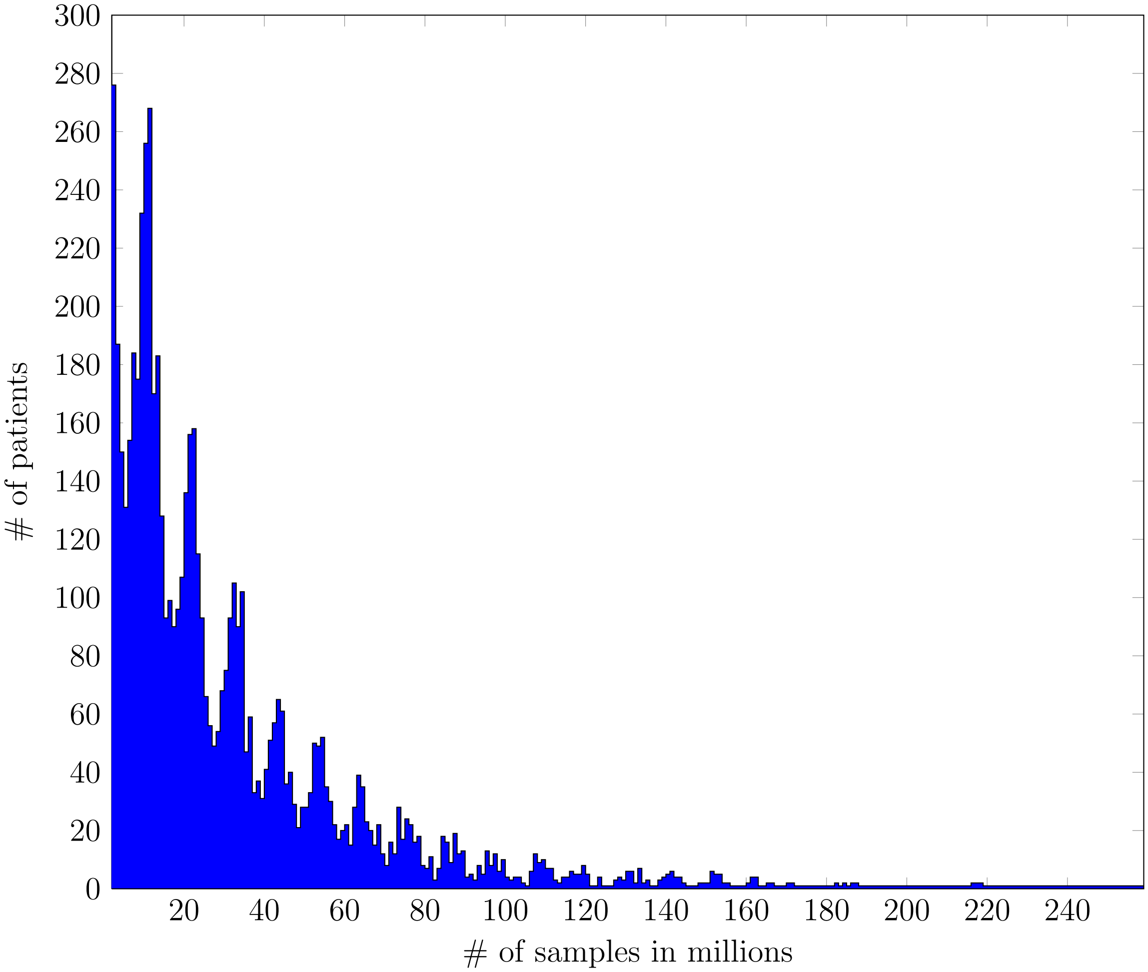

Figure 2-6: A graph showing the number of samples in each signal file on MIMIC.

times larger than the average size. Processing all of this data and extracting features into memory requires a significant amount of RAM. There is a tradeoff between ease and simplicity of implementation and the ultimate cost of running anaylses using PhysioMiner.

The amount of RAM necessary changes the costs of running an analysis, as EC2 server costs scale with the instance specifications. Table 2.1 shows a comparison of RAM, CPU, and cost for various EC2 instances. There is a cost tradeoff between how fast computation can be performed, and how much memory the instance has. Memory is particularly important, since Java garbage collection tends to take a significant amount of time when the heap size gets too close to the maximum. Ultimately, the

R3 family of EC2 instances was a better bang for the buck. Using a one-size-fits all approach, we would need to use a `R3.2xlarge` instance to support the largest files. Using the `R3.2xlarge` would cost $0.70 per hour per instance, which could make costs of an analysis fairly high. There are two possible solutions to reduce the costs: 1) Divide up tasks by file size, and 2) reduce the memory footprint via partitioning.

**1) Divide tasks by file size**

Since not all of the files are the same size, different size instances can be used to process different size files. If 50% of the files are less than half the size of the largest file, then the this method would save 25% on server costs. Realistically, more than 80% of the files tend to be much smaller than the largest file, so the savings in server costs would be substantial.

This requires a lot more configuration and implementation, as instances need to be aware of their specifications, and there needs to be a separate queue for each type of instance. This will eventually be a built in feature of PhysioMiner, but it is possible to do this manually by using the populate module multiple times.

**2) Partition files into memory**

Currently, the memory requirements are a function of the number of samples in the signal file. This means that, as samples are stored at higher resolutions and for longer time periods, memory requirements will go up. This is not strictly necessary, as most of the modules only require a small portion of the signal at any given time.

Files could be split up into chunks, which are each loaded and processed separately. This would require some amount of care, as often beat features depend on previous beats. In addition, most open source onset detection algorithms expect a full signal file, so stitching the results together would require some work. Though this is out of the scope of the original implementation of PhysioMiner, this is definitely possible and would decrease the server costs of running an analysis.

### 2.3.4 Scalability: Database Interactions

Initial tests of PhysioMiner showed that the majority of the populating time was not from data processing - it was from time spent writing to the database. Amazon DynamoDB allows users to pay for more read and write throughput. To provision an additional 10 writes per second costs \$0.0065 per hour.

A scaling test was run using the populate module with 5 patients and 200 writes per second. 80% of the time was spent writing to the database. Using 800 writes per second reduced the database write time to about 50%. For an hour of analysis, this amount of throughput would cost \$0.52. While this seems like a trivial amount, it scales with the amount of data to analyze. To analyze 1000 patients with one million beats each in 5 hours, this would require more than 50,000 writes per second. Over five hours, this would cost \$162.50. These costs scale linearly with the amount of data to process. PhysioMiner is scalable for large data sets, but scaling is costly. However, the creation of beatDB which uses the populate module is only done once per new data source.

Unfortunately, there are no solutions to this problem other than using another database solution. There is no time/cost tradeoff here - increasing the allowed time does not change the total cost of database throughput. Amazon has a concept of reserved capacity - purchasing throughput for an entire year at a steeply discounted price. However, since most analyses are done sporadically, this would not be a good solution for the typical user of PhysioMiner.

## 2.4 Other Considerations

### 2.4.1 Cloud Services

There were several options considered when deciding what cloud framework to use for PhysioMiner. We considered using both OpenStack and Amazon. Ultimately, we decided to use Amazon because of the variety of services it had that could be easily integrated into PhysioMiner. In particular, services like Amazon SQS were critical in making PhysioMiner a simple and scalable solution.

We were initially deciding between using Amazon Relational Database Service (RDS) and Amazon DynamoDB. RDS is a SQL database designed to be fast, though it has scalability issues with data sizes over 3TB [2]. With the following considerations, we decided that DynamoDB was a better solution for PhysioMiner:

- NoSQL databases tend to perform better when there are few JOIN operations. The workflow that PhysioMiner uses requires no JOIN operations.
- DynamoDB provides a much better scaling framework. Users can pay for more throughput to the database, with extremely high limits.
- RDS is currently limited to 3TB per instance. If the amount of data grows larger than this, then the user will have to implement a sharding technique for their data. The sizes of tables in PhysioMiner are expected to grow to terabytes in size, so this limit would become a problem.

### 2.4.2 File Formats

PhysioMiner has built in support for ABP and ECG signal files stored in EDF format. The public MIMIC database of signal files contains patient data stored in MIT format. Each patient is stored as a group of header files and data files, corresponding to segments of time where signals were continuously recorded. This format is interchangeable with the EDF format, using the Physionet software *mit2edf* [8]. The decision to support EDF over MIT was somewhat arbitrary, but since the EDF format is easier to work with (a single file), we selected EDF.

# Chapter 3

# Building Predictive Models

PhysioMiner provides a framework to build predictive models over the feature repository data. At a very high level, this framework allows users to specify events and conditions that occur on a patient's time line, and build a classifier on the features in the feature repository to predict the events ahead of time. PhysioMiner provides an interface to specify conditions, and features to use for the classifier. It will eventually have a built in classification system, but it currently ports them into Matlab or Delphi.

Hypothesis testing is split into two separate modules in PhysioMiner- **Conditioning** and **Aggregating**. These modules can be used separately or together to extract the aggregated features for each event. The basic structure of feature aggregation is illustrated in Figure 3-1.

## 3.1 Condition Module

Users can find all windows matching a specific condition using this module. Medical events are often directly related to a specific condition of the physiological wave form, and finding such windows allows users to find these events. For example, *acute hypotensive events* can be defined in terms of the mean arterial pressure over a given

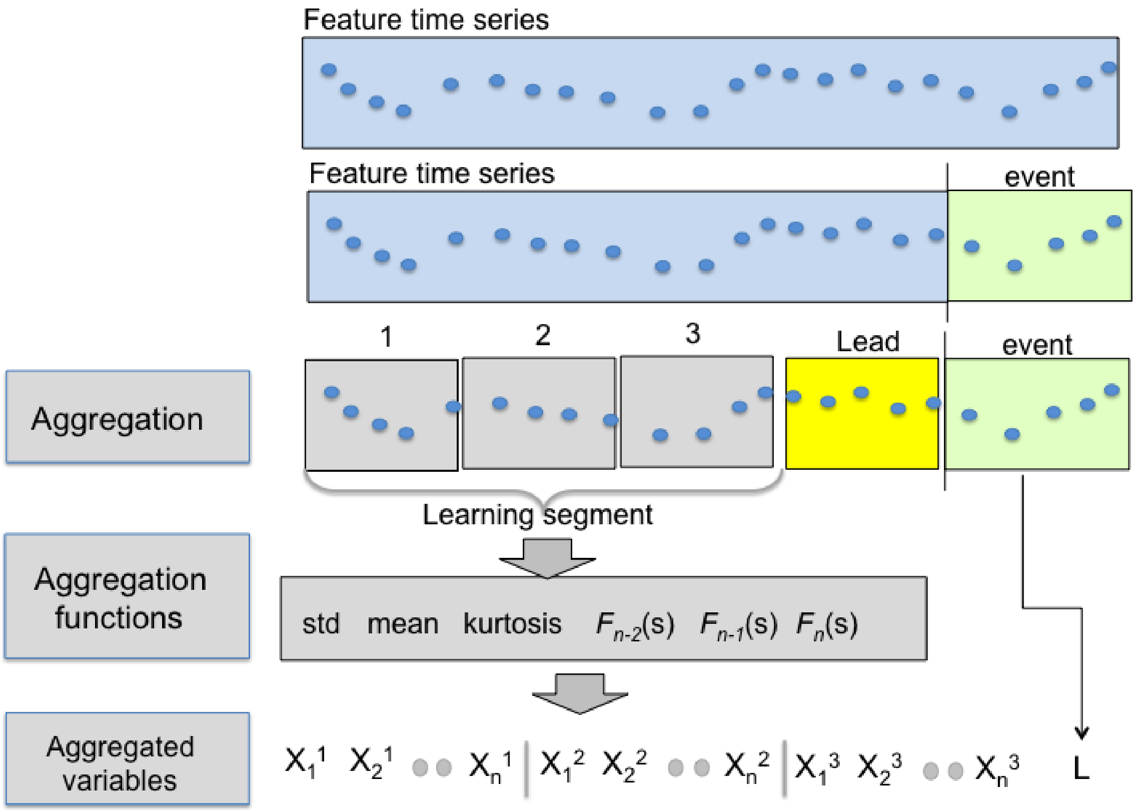

Figure 3-1: The basic workflow of feature time series aggregation. A window is specified corresponding to a patient event. A time period before that is designated the predictive window, or learning segment. This window is divided into subwindows, and feature time series is aggregated for each of these windows. For $n$ aggregation functions and $m$ beat feature time series, there are $nm$ aggregated covariates for machine learning per subwindow.

duration. The overall structure of the Condition module is illustrated in Figure 3-2.

### User Specifications

PhysioMiner only currently supports built in conditions, but will soon support arbitrary condition scripts. The user provides:

- S3 Bucket - container in S3 with the scanning script
- Condition script location - file path in S3

Optionally, aggregators and feature names can be given to the populate module to compute them at initialization time. This allows PhysioMiner to be both flexible and

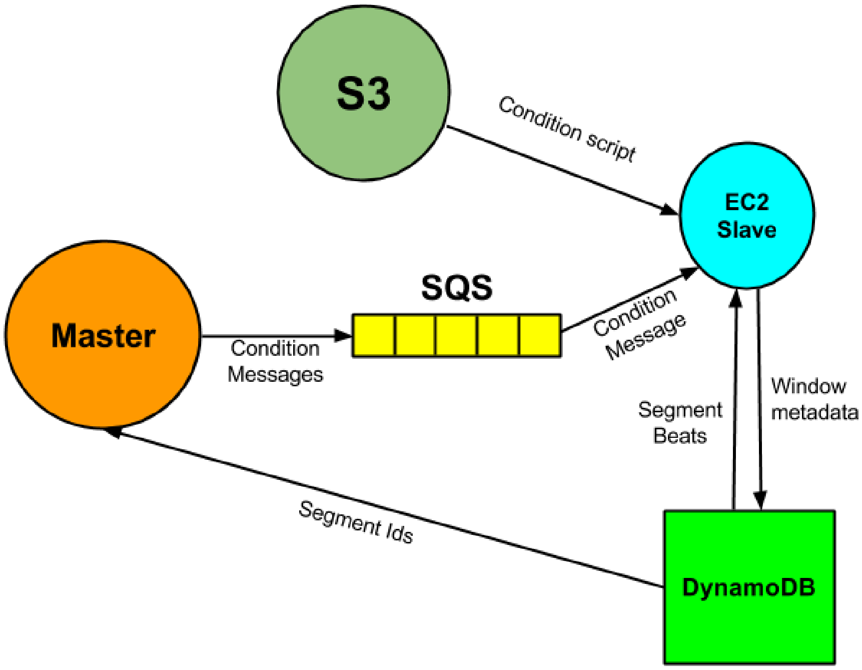

Figure 3-2: The framework that the Condition Module uses. Each message corresponds to a segment to process. The slave reads the beat metadata from the database, finds any windows that match the specified condition, and stores them into the appropriate window table.

economical by minimizing the data processing time.

### Master

The master retrieves a list of segment ids from the database. It then creates a `ConditionMessage` for each segment, and pushes it to Amazon SQS. A sample `ConditionMessage` might look like:

```
S3 Bucket: mybucket
Segment Id: 7hf73hYAU83AnnHCBAL
Condition Script: scripts/acute_hypotensive_episode
Window Table Name: AHE
```

If aggregators were given to the master, then the `ConditionMessage` also contains a list of feature names, and a list of aggregators.

**Slave**

Once a slave receives a `ConditionMessage`, it performs the following steps:

1. Downloads the beats and features for the given segment
2. Searches the segment for a window that matches the given condition
3. Adds all matching windows to the appropriate window table

**Customization**

Currently, users can only use built in condition scripts. PhysioMiner will eventually be able to support arbitrary condition scripts to determine windows.

However, PhysioMiner allows users to input windows directly into the database. If the user has a list of predefined event times, they can write those windows into the database, and use the Aggregate module described in the next section.

## 3.2 Aggregate Module

The main use case of PhysioMiner is using machine learning to predict certain physiological events. Users can aggregate feature time series from their conditioned windows using this module, and use these as input covariates into a classifier (logistic regression, SVM, etc). PhysioMiner will soon support subaggregation windows, which divides the predictive window into several chunks before aggregating. The overall structure of the Condition module is illustrated in Figure 3-3.

**User Specifications**

The user provides:

- **Window table name** - the table in DynamoDB containing all windows to aggregate features for

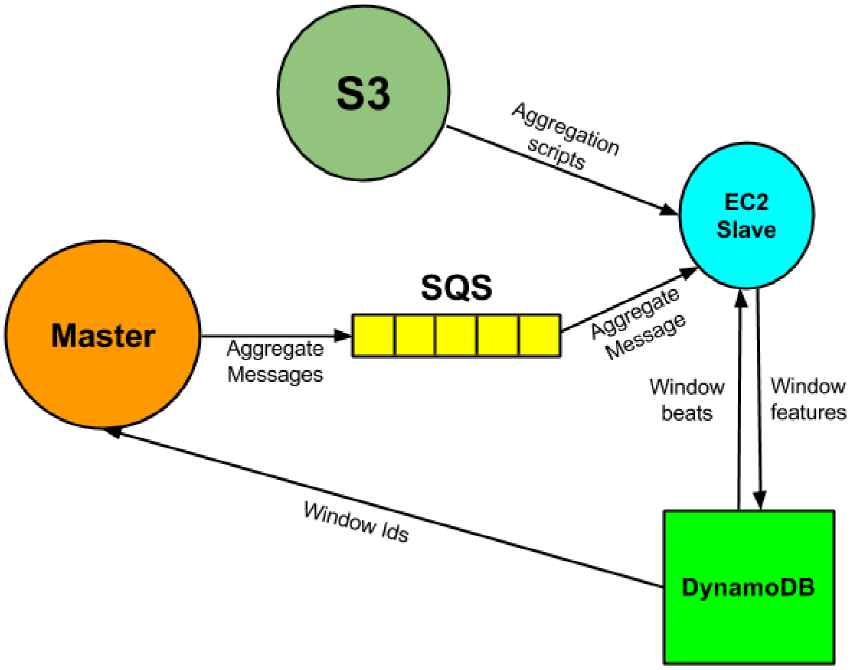

Figure 3-3: The framework that the Aggregate Module uses. Each message corresponds to a window to process. The slave reads the window and beat values from the database, aggregates features, and stores the features back into the database.

- **List of feature names** - the features to aggregate from these windows
- **List of aggregation functions** - the aggregation functions used to combine each feature (mean, skew, etc)
- **Lag** - the length of history to aggregate features on
- **Lead** - time interval between the last data point of history and the beginning of the window
- **Subaggregation Window** - time interval to divide windows into before aggregating

Lead and lag are illustrated in Figure 3-4.

PhysioMiner currently has built in support for five different aggregation functions: **mean**, **standard deviation**, **kurtosis**, **skew**, and **trend**. PhysioMiner will soon

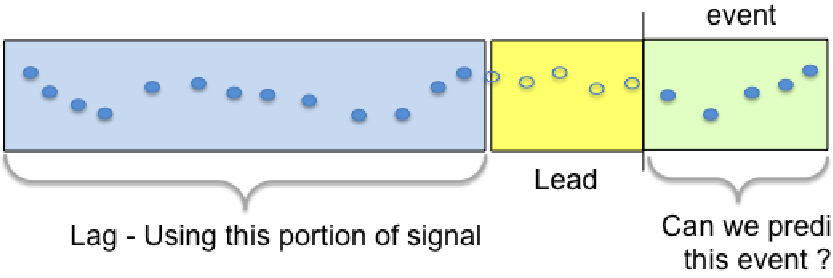

Figure 3-4: An window is defined by some interval corresponding to a patient event. The lead is the gap between the predictive window and the event. The lag is the duration of the predictive window.

support arbitrary aggregation scripts which the user can provide. These scripts would take in a list of feature values, and return a single aggregated value. A typical use case would look like:

```
>>> ./aggregator 57.2 43.2 34.1 56.7 91.1 22.2
    53.54
```

### Master

The master retrieves a list of windows from the window table that the user specified. It then creates an `AggregateMessage` for each window, and pushes it to Amazon SQS. A sample `AggregateMessage` might look like:

```
Window id: HUje8kwmqoN839Gha0
Feature names: duration, heart_rate
Aggregator Functions: mean, skew, trend, kurtosis
Lag: 1800
Lead: 600
Subaggregation Window: 60
```

### Slave

Once a slave receives an `AggregateMessage`, it performs the following steps:

1. Reads all feature time series for the segment in the predictive window
2. Divide the window into subaggregation windows of the appropriate size

3. Perform each aggregation function for each given feature time series
4. Store all aggregated features into the given window table

**Customization**

Users will be able to specify arbitrary aggregation functions to use, instead of just the ones built into PhysioMiner. Since, the typical number of features and aggregators will be small, these scripts can be arbitrary binary executables, without suffering a big performance hit. PhysioMiner does not yet support arbitrary aggregation functions, but this is left as future work.

# Chapter 4

# New Patients and Features

One of the goals of PhysioMiner was to be flexible - to allow users to add patients, add features, and create new databases. The modules described in the previous two chapters allow this flexibility.

## 4.1 Adding Patients

When populating the database, the user has the option of recreating the tables. If the user elects to recreate the tables, then all existing data is wiped, and the database will consist only of the patients the user just added. If the user elects not to recreate the tables, then patients are simply added to the existing database. PhysioMiner uses the same framework for initialization and adding patients, so this process is seamless to the user. The user justs specify which features to extract from the new patients during populating.

## 4.2 Adding Features

PhysioMiner's feature extract module allows users to easily add features at any time. While users will typically extract most features during the initial populating of the database, it is just as easy to extract features for existing patients. The process of extracting features is very similar to populating the database – the user specifies the table name, patient folder, and features to extract. The feature extract module does not require an onset detector, since all onsets have already been calculated and stored in the database

## 4.3 Adding Signals

PhysioMiner does not directly support multiple signals. Multiple signals for the same patient would divide into different onsets, so storing them in the same database would be complicated. Instead, PhysioMiner allows the user to specify the table name everytime they run a PhysioMiner module. This allows users to manage multiple databases with different signals. If the patient names are the same, it is easy to write a script to link two patients across tables.

# Chapter 5

# Demonstrating with ECG Data

To demonstrate PhysioMiner's interface firsthand, we extracted several features from ECG data stored in the online public MIMIC database. [1]. We selected 6933 patients with ECG Line II data to extract features from. We then used already-known acute hypotensive events to test the analytics framework of PhysioMiner. Listed here is a detailed analysis of the steps taken in extracting these features, the problems that occurred along the way, and the various solutions to these problems.

## 5.1 Selecting the Patients

Physionet provides a variety of tools to interact with their signal files, without downloading the file from the database. In particular, `wfdbdesc` is a program that gives a summary of a patients file - which signals are included, the sampling rate, the duration, etc. [17]. Using this, we were able to iterate through all the patient records in the MIMIC II Waveform Database, and determine which signals they contained. We selected all patients that had both ABP and ECG Line II data.

---

[1] We picked the same patients for whom the ABP features were extracted in the original BeatDB paper [18].

## 5.2 Downloading the Data

PhysioMiner requires that the patient files be located in Amazon S3. Since the original data is stored in the MIMIC database on Physionet, we had to download the data as an EDF file, and store it in S3. There were two possible workflows that were considered:

1. Use mit2edf to download the file directly as an EDF, then upload the EDF file to S3.
2. Download the file as a zip, convert it to EDF, then upload the EDF file to S3.

Even though the EDF and zip file sizes were similar, the second workflow was about 20 times faster. Though the reasons are unclear, it is likely that the conversion from the MIT file format to EDF file format is extremely slow on the MIMIC servers. We implemented the second workflow to download all of the patient data.

The MIMIC II Waveform Database is split up into 10 separate databases, each comprising approximately one tenth of the patients. This allowed us to easily parallelize the download process by doing each database independently. We used the following steps:

1. Create an EC2 instance with a 200GB EBS volume for each of the 10 databases.
2. Use rsync to download the entire database onto the EC2 instance.
   ```
   rsync -Cavz physionet.org::mimic2wdb-30 /usr/database
   ```
3. Run a python script to convert each file to EDF format, and upload it to S3.

This process took about 15 hours. Instead of uploading all of the patient files into the same folder in S3, they were split up by total number of samples in the file. This allowed us to have more fine grained control over speed and costs, which will be detailed in Section 5.3.3.

## 5.3 Populating the Database

This was the first large scale test of PhysioMiner. Our goal was to do onset detection and extract a large number of features from the ECG Line II data while populating the database, to minimize time and costs. Populating the database required both a sophisticated onset detection algorithm, and feature extraction scripts. We implemented both of these.

### 5.3.1 Onset Detection

PhysioMiner requires an onset detection script to divide a signal file into beats. There is no accepted way to divide ECG data into beats, so we used the structure of a typical ECG waveform to divide it into beats [19].

A typical waveform is shown in Figure 5-1. There are distinct points along the wave labeled P, Q, R, S, and T. While some beats may only have a subset of these, all beats will contain a QRS complex. Thus, we used a QRS detection algorithm to find the QRS complexes, and subsequently divided the signal into beats around these complexes.

Given the QRS complexes for a given signal file, we simply create beat divisions in between each complex. This gives us a beat roughly centered around each QRS complex. This can be seen in Figure 5-2.

To calculate the start of the beat, we use the following procedure:

1. Start at the left side of the QRS complex (the Q).
2. Travel left until you reach either a halfway boundary, or a gap.
3. Mark this as the start index

To calculate the end of the beat, we use the following procedure:

1. Start at the left side of the QRS complex (the Q).

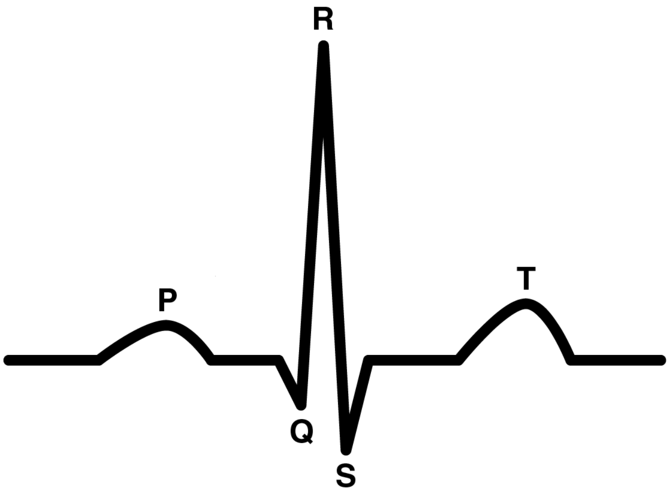

Figure 5-1: A typical ECG waveform. It can be divided into a P wave, a QRS complex, and a T wave. The QRS complex is always present, while the outside waves are sometimes hard to locate. [15]

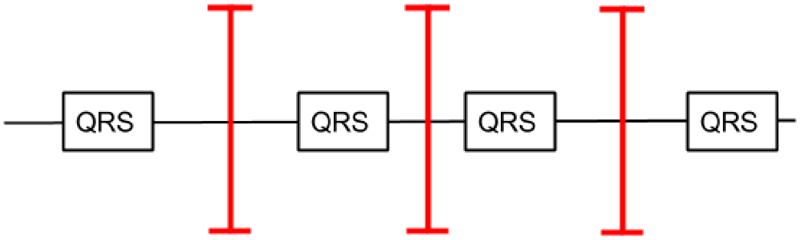

Figure 5-2: The division of an ECG waveform into beats. The start of each beat is exactly halfway between consecutive QRS complexes.

2. Travel left until you reach either a halfway boundary, or a gap.
3. Mark this as the start index

A specific example of using this algorithm is shown in Figure 5-3.

To detect QRS complexes, we used the `wqrs` software listed by Physionet (cite). This software annotates the beginning and ends of each QRS complex in a signal file, even when that signal file contains gaps.

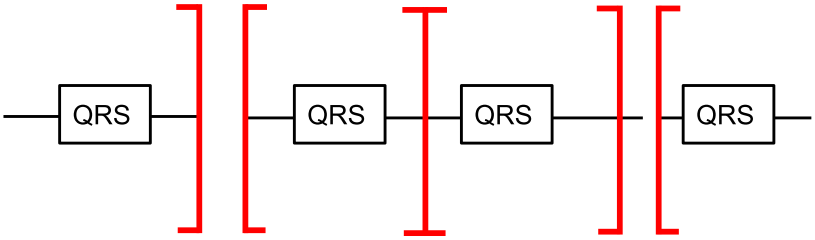

Figure 5-3: The division of an ECG waveform into beats, using the specified algorithm. Notice that for the last boundary, there is some data not included in any beat

### Onset Detection Challenges

After implementing the above onset detection algorithm, we ran it on several patients to see how it performed. We noticed odd behavior start to occur at the end of files. The `wqrs` detection program began detecting QRS complexes that were within 200ms of each other. This would correspond to a heart rate of over 300bpm.

After plotting the waveform, and the calculated QRS complexes, we found the issue. The `wqrs` program was mistakenly identifying P-waves as QRS complexes, as shown in Figure 5-4. After experimenting with the software, we found that the results became more accurate when the analyzed signal was short. The `wqrs` program is a stateful QRS detector, so previous QRS complexes affect the detection of future ones. If the signal file contains a particularly bad section of data, the detection can often get skewed.

To solve this issue, we divided the signal file into chunks, and detected QRS complexes separately. Each chunk had a small overlap, and we post-processed the data to remove duplicates. If the chunk size was 100, and the overlap was 10%, then the chunks would be from 0 - 100, 90 - 190, 180 - 280, etc. This reduced the propagation of errors in the QRS detection, and resulted in significantly less false positives. The results from each chunk are sorted and concatenated together. To remove duplicates, we traverse the concatenated list and remove any onsets that are not in sorted order. This ensured that there were no overlapping onsets, but also that we correctly did onset detection over the entire signal. There was a tradeoff between the chunk size, and the total

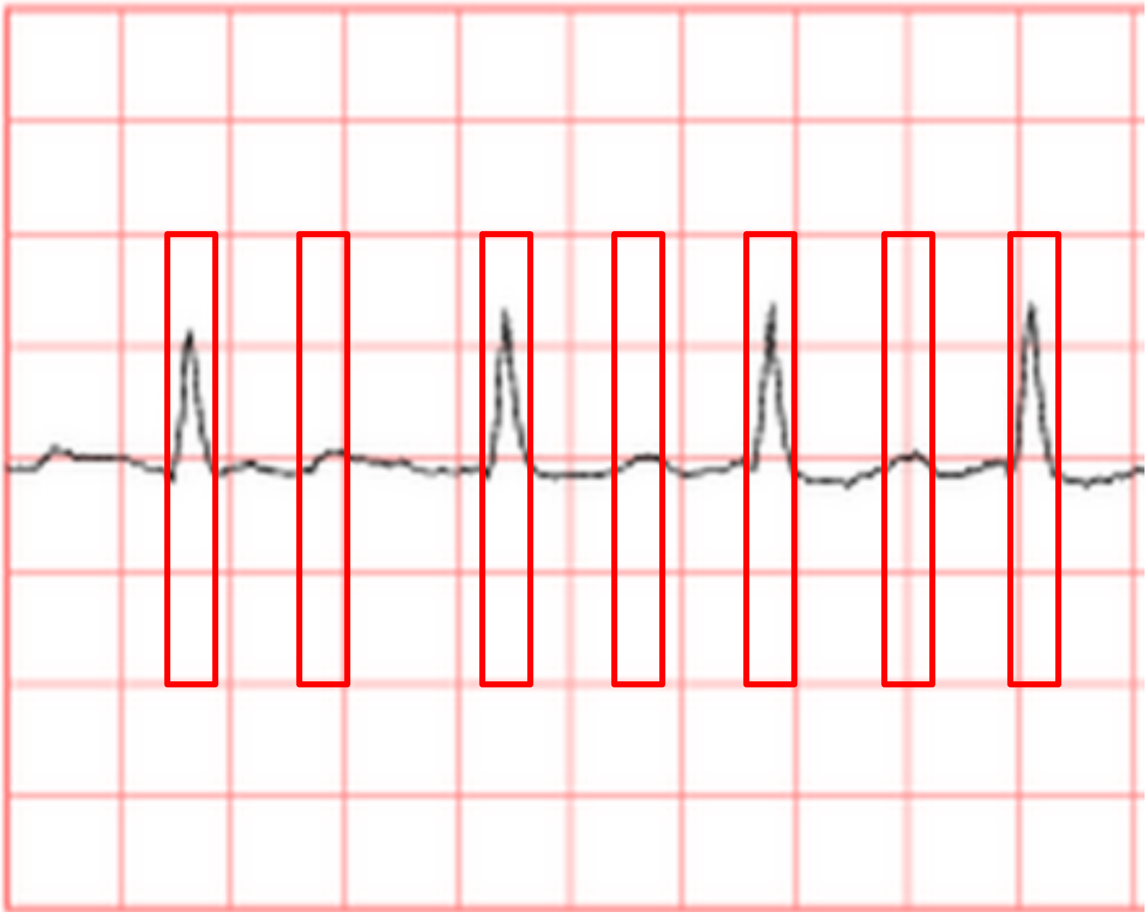

Figure 5-4: The `wqrs` software sometimes mistakenly identifies P waves as QRS complexes. The red boxes indicate where the `wqrs` software indicated a QRS complex. However, only every other box has a real QRS complex, as indicated by its large peak.

running time of the onset detection. We settled on using chunk sizes of 100,000 and an overlap of 10%.

### 5.3.2 Feature extraction

A lot of research has gone into extracting features from ECG data. These features come in several forms - *wavelets*, *intervals*, and *morphology*. Together, these give an accurate summary of a beat. The extracted features are listed below, by category.

### Wavelet Features

A wavelet transform takes a signal, and returns an output with both frequency and location information. There are several types of wavelet families that can be used. We found that the Daubechies 4 and Symlet 4 wavelet family were used with success in ECG feature extraction so we used coefficients from each of those [21].

- **Daubechies 4** - the discrete wavelet transform of the beat using the Daubechies 4 wavelet family. We extracted the first four coefficients of the approximation vector and first four coefficients of the detail vector.
- **Symlets 4** - the discrete wavelet transform of the beat using the Symlets 4 wavelet family. We extracted the first four coefficients of the approximation vector and first four coefficients of the detail vector

### Interval Features

We stored various interval related features, which are illustrated in Figure 5-5.

- **RR Interval** - the time interval in between consecutive R points (saved on the subsequent beat)
- **Heart rate** - the inverse of the RR Interval
- **QS length** - the time interval between the Q and S points
- **QR length** - the time interval between the Q and R points
- **RS length** - the time interval between the R and S points
- **Q start** - the interval between the start of the beat and the Q point

### Morphological Features

- **Dynamic Time Warping** - a measure of the similarity between consecutive beats, that accounts for changing time and speed (saved on the subsequent beat)

We used the Python PyWavelets package to do the discrete wavelet transform [13], and the Python mlpy package to do dynamic time warping [16].

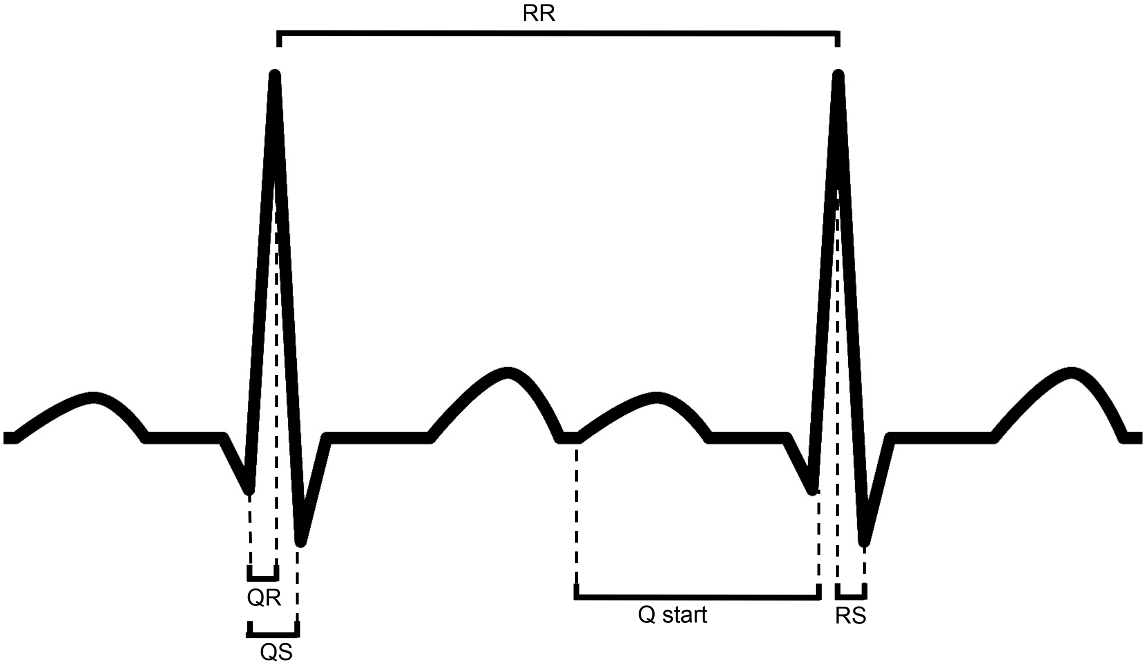

Figure 5-5: The `wqrs` software sometimes mistakenly identifies P waves as QRS complexes. The red boxes indicate where the `wqrs` software indicated a QRS complex. However, only every other box has a real QRS complex, as indicated by its large peak.

**Feature extraction challenges**

We ran the feature extraction scripts on a few patient files to make sure it worked as expected. We ran into two unexpected issues, both of which were easily resolved.

The first issue was with dynamic time warping (DTW). Occassionally, the mlpy package would throw a MemoryError during the calculation of the DTW. With some experimentation, this occurred when the two input arrays were larger than 50,000 elements each. A typical beat only has around 100 samples in it, so this case only occurred when the data was extremely noisy (or nonexistent). In particular, this happened several times when the data was all zeros for several minutes. To fix this issue, we simply ignored any beats which were longer than 1000 samples while doing dynamic time warping. These beats were already invalid, and would not count towards in the aggregation functions anyway.

The second issue was with the discrete wavelet transforms. Occasionally, the detail and approximation vectors had less than 4 coefficients in them. This occurred when the input arrays were less than 4 samples long. These corresponded to invalid beats (when the QRS detector had too many false positives), so we simply ignored these beats while doing wavelet transforms.

### 5.3.3 Challenges at Scale

There were several issues that arose when running PhysioMiner at scale that did not occur in initial small scale testing. These issues, and their solutions, are documented here.

**Scaling costs**

One of the goals of PhysioMiner was to make it economical. To analyze the projected costs of this analysis, we used the following considerations:

- R3.2XLarge EC2 Instances - we need instances this large to support the larger files
- 100,000 writes/sec throughput to the database
- An average signal file has 300,000 beats
- Processing an average file (excluding database writes) takes about 4 minutes

Using these assumptions, we estimated to have a total of 2.1 billion beats to write to the database. At the given throughput, this would take 21000 seconds, or 5.83 hours. This means, the total processing time would be about 11 hours with 100 EC2 instances. The total DynamoDB costs would be 11 hours * 100,000 writes/sec * 0.00065 =715. The total EC2 costs would be 1 hours * $0.70/hour/instance * 100 instances = $770. This brings the total cost to $1485. This did not align with our goals of being economical.

To mitigate this, we divided our signal files by number of samples. This allowed us to use smaller EC2 instances where needed. More than 4000 of the patients could be run using the R3.Large instance, which costs only $0.175/hour. This would reduce the

EC2 costs by about $450, bringing the total to about $1000. We ran the Populate module several times, using different instance sizes each time. We ended up not analyzing the 800 largest patients. The total cost ended up being about $1000.

### Database Throughput

DynamoDB has a pay-per-throughput payment model, which allows users to pay for exactly how much throughput they need. However, this is the maximum throughput allotted to the database - it is divided amongst various partitions. Since partitions are by primary key, Amazon suggests structuring the database such that primary keys are approximately uniformly accessed at any given time.

Unfortunately, this structure does not work well with PhysioMiner. We need to group beats by segment id, and we need to write all of the beats of a given segment id at the same time. This means that a partition may have a heavy load at any given time. Using 20,000 writes/sec theoretical throughput for the database, we were only able to achieve 5000 writes/sec actual throughput using 20 instances.

The more instances we can run at a time, the more spread out the database writes will be across primary keys. Using 50 instances instead of 20, the actual throughput jumped from 5,000 to 10,000. We requested Amazon to increase the user limit on EC2 instances for us so we could maximize throughput to the database.

### Failing Instances

Throughout the initial run, we found and fixed several of the problems that were listed above. However, each time one of these problems occurred on the slave EC2 instance, the slave program would crash. PhysioMiner was built to handle these sort of failures - if a slave crashed, its task would eventually be put back into the queue for another slave to handle.

However, there was no way of knowing which nodes had failed, besides manually watching all of them. To solve this, we implemented a simple Keepalive protocol to check on the slaves. Every few seconds, each slave would post a message to the KeepAlive Queue in Amazon SQS. The master would read these messages as they came in, and keep track of who was alive. If the master did not hear from a slave in more than 30 seconds, it informed the user that one of the slaves had died.

This solved two issues. First, it pointed us directly to the problem slave, so we could find and solve the problem that arose. Then, we could simply restart the slave program on that instance. Second, it saved us EC2 costs by not keeping around failed instances. In the initial run through, more than half the instances failed with various issues that were eventually fixed. This would have been a serious cost burden without the notification system.

### No Transactions

Failing instances caused more problems than just restarting the slave. Often times, these slaves would fail in the middle of writing to the database. Since segment ids were randomly generated, simply rerunning the task would not be idempotent. In addition, the task would get added to the queue again, so it would likely run on another slave and have duplicate data in the database.

To fix this issue, we noted down the patients which caused problems, and scanned the database for them at the end. If any of the patients had multiple segments where we were only expecting one, we deleted both from the database. We then did a second run to repopulate the database with these patients. The total number of problem patients was small, so this did not add any significant time or cost to the processing.

This could have been solved in multiple ways. The slaves had multiple bugs that were fixed throughout the initial initialization process. After fixing the bugs, there were no more instance failures. It turns out that instance failures were not that apparent in practice, and were only a result of a few small bugs.

However, for large scale analyses, we need PhysioMiner to be completely fault tolerant - even if the instance itself crashes, the database should be intact. Unfortunately, DynamoDB does not support transactions, so this is much harder. The easiest way to accommodate this would be to make all tasks idempotent. This way, running a task twice (or one and a half times) would result in the same output to the database. We could accomplish this by derandomizing segment ids. DynamoDB does not provide an incremental id on primary keys, so this would require a decent amount of engineering. This is probably the best solution, however, and will be left as remaining work.

## 5.4 Building Predictive Models for AHE

After populating the database, the next step was aggregating features. We had a predetermined list of timestamps and windows where AHE events occurred in the list of patients. After converting these time stamps to beat ids, we wrote a simple script to populate a `window` table with the given intervals. We then used PhysioMiner's aggregation module to collect features.

We did not have a list of non AHE events, so these were generated instead. We selected time intervals that were at least 5 hours away from an AHE event, and added it to the window table with a n̈ormalc̈lassification. Our final window table had 3323 AHE windows and 476 normal windows.

For each feature listed in Section 5.3.2, we calculated five different aggregations: mean, standard deviation, skew, kurtosis, and trend. We collected 23 features per beat, so there were 115 total aggregated features for each window. We ran this module for a lead of 10 minutes and a lag of 30 minutes. Running on a single EC2 instance and using 6000 writes/sec throughput to the database, this process took 25 minutes to complete.

### 5.4.1 Building a Classifier

PhysioMiner does not currently have a classification algorithm built into it. However, Amazon provides a mechanism to export the database into a text file using Amazon Elastic Map Reduce (EMR). After exporting the database to text format, we converted it to CSV and ran it through logistic regression. The results were (unsurprisingly) not very accurate. With a lead of 10 minutes and lag of 30 minutes, we achieved an area under the curve (AUC) of the receiver-operating characteristic curve (ROC) of 0.57.

We then used Delphi, a multialgorithm, multiparameter, distributed meta-machine learning platform that is currently in beta [6]. It found that the best machine learning algorithm for the given data was an SVM using radial basis functions. This achieved an AUC of 0.693 on cross-validation testing. The various ROC curves produced by Delphi are plotted in Figure

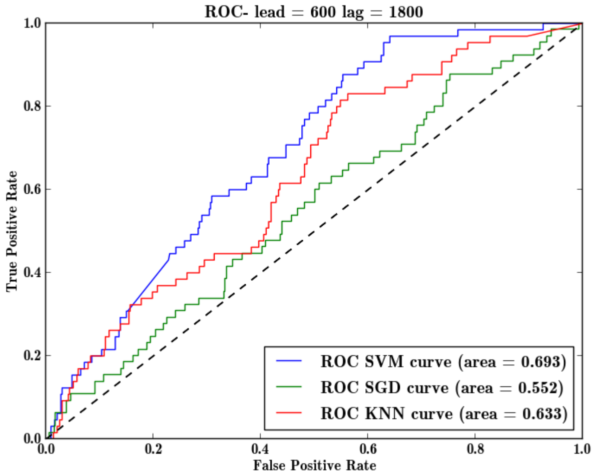

Figure 5-6: Three different ROC curves produced by Delphi for different machine learning algorithms.

While this is a reasonable area under the curve, there are several reasons why it might not be as high as possible:

- ECG values are not good predictors of AHE events. Acute Hypotensive Episodes are based on blood pressure, so it seems reasonable that blood pressure signals would be better predictors of AHE than ECG signals.
- We generated the non-AHE events based on a list of AHE events, but it is possible that the time periods we chose were not normal. The final predictor based on logistic regression classified 95% of the training data as AHE, and inaccurate classifications would explain that.
- The onset detection for ECG was challenging and could have lead to inaccuracies. The `wqrs` software used to find QRS complexes often mistakenly identified

P waves as QRS complexes. This resulted in a heart rate that was double the actual heart rate of the patient.

## 5.5 Summary of Results

| | |
|---|---|
| **Total number of patients** | 6118 |
| **Total number of beats** | 1,138,953,214 |
| **Total number of Features extracted** | 26,195,923,922 |
| **Total size of database** | 504 GB |
| **Total time taken** | 20 hours |
| **Number of nodes used** | 60 |
| **Type of node used** | `R3.Large` |
| **DynamoDB throughput used** | 30,000 writes/sec |
| **AUC for ROC curve** | 0.693 |

# Chapter 6

# Conclusion

We have presented PhysioMiner, a large scale machine learning and analytics framework built in the cloud. PhysioMiner is built to handle terabytes of physiological signal data, allowing users to clean, process, extract features, and analyze the data at scale. Using just four simple modules based on the master-slave framework, PhysioMiner is able achieve flexibility and scalability, without sacrificing accuracy and reliability.

## 6.1 Future Work

There is a lot of work left to improve upon for PhysioMiner. Many of these issues have been documented in the sections above, but we will list them here again:

**Arbitrary Feature Extraction**

While PhysioMiner supports binary executables for feature extraction, these have a tremendous cost in both time and money. There are two possible solutions. One is to figure out a way to support binary executables without sacrificing speed. The other is to add support for other languages. Currently, feature extraction support is only written for Python.

**Arbitrary Aggregation Functions**

PhysioMiner currently only supports five built in aggregation functions. To be completely flexible, it should allow users to specify aggregation functions via binary executables. These functions would simply take in a list of feature values, and output a single aggregated value.

**Memory management**

PhysioMiner requires RAM proportional to the size of the signal file. With some care, PhysioMiner could split up segments into constant size regions, and process them separately. This would bring the RAM requirements down to a constant size, allowing users to use the same size instance for all jobs and still be cost effective.

**Window conditions**

Curruently, PhysioMiner only supports built in window conditions. Since it is open source, users can directly write their window conditions into the source. However, it would be ideal if these conditions could be written as separate modules, and uploaded to S3 before the start of an analysis.

**Machine Learning**

PhysioMiner does not have a built-in machine learning framework. However, it easily allows users to export the data from DynamoDB, and run their own machine learning scripts. This process should be pipelined so that users can easily see the results of their classification models.

## 6.2 Future Goals

We hope that PhysioMiner allows researchers and practitioners to focus on analyzing their data, and not building and maintaining a cloud system. We have designed it to be flexible, general, and scalable so that it reaches the maximum number of users. We hope that with such a system, users will find it easier to process and analyze their data. We hope that this ultimately stimulates new research and results.

# Appendix A

# PhysioMiner Interface

This chapter will discuss exactly how a user interacts with PhysioMiner. This should be used as a guide to how PhysioMiner works, but for up to date documentation, look at `https://github.com/gigabeats/beatdb/wiki/Tutorial`.

Users should start by downloading the JAR files from `https://github.com/gigabeats/beatdb/`. They will need JRE 7 and an AWS account to get started. Command line options can be found by running:

```
java -jar physiominer.jar --help
```

## A.1 Populate

PhysioMiner Populate accepts the following command line arguments:

```
-a, --ami <arg>           Amazon Machine Image to launch instances
                          with (default is the public Physiominer AMI)
-b, --bucket <arg>        S3 Bucket
-d, --folder <arg>        S3 data folder containing patient signal files
-f, --features <arg>      File containing feature extraction scripts
                          This file should contain a row for each feature
```

```
                            in the format "<feature_name> <script_path_in_S3>"
 -i, --initialize           Pass this argument to initialize the tables for
                            the first time. If tables with the given table
                            name already exist, they will be deleted and
                            recreated
 -n, --num-instances <arg>  Number of instances to create
 -o, --onset-detector <arg> Location in S3 of the onset detector algorithm
 -p, --iamprofile <arg>     Name of IamProfile to use for EC2 instances
                            (must have access to DynamoDB, S3, and SQS).
 -s, --signal-type <arg>    Optional, the signal type
                            (possibilities: ECG, ABP). If a signal type is
                            specified, a built in onset detection algorithm
                            is used.
 -t, --table <arg>          Name of table to store in the database (names will
                            be "<table_name>_beats" and "<table_name>_segment")
 -y, --instance-type <arg>  Amazon EC2 Instance type to launch
                            (possibilities: r3.large, r3.xlarge,
                                          r3.2xlarge, r3.4xlarge, r3.8xlarge,
                                          c3.large, c3.xlarge, c3.2xlarge,
                                          c3.4xlarge, c3.8xlarge)
```

To populate the database, the command to run might look like:

```
java -jar populate.jar --bucket mybucket --folder mydata
   --features scripts/ecg/features.txt --num-instances 3
   --iamprofile myiamprofile --table ecg --instance-type r3.large
   --signal-type ECG --onset-detector s3path/to/onset_detector
```

Running the above line will create 3 `r3.large` instances using the public PhysioMiner AMI. It will create an entry in the `ecg_segments` table for each patient the the mydata folder. It will use the built in ECG onset detection to split each signal file into beats. It will download all feature scripts listed in features.txt, and calculate these features for each beat. Each beat will be stored as a row in the `ecg_beats` table.

## A.2 Extracting Features

PhysioMiner `FeatureExtract` extracts features for existing patients in the database. It accepts the same arguments as Populate, with the exception of the onset detector. All onsets have already been calculated, so this is no longer necessary. Running feature extraction might look like:

```
java -jar feature_extract.jar --bucket mybucket --folder mydata
   --features scripts/ecg/features.txt --num-instances 3
   --iamprofile myiamprofile --table ecg --instance-type r3.large
   --signal-type ECG
```

Running the above line will create 3 `r3.large` instances using the public PhysioMiner AMI. For each patient in the `ecg_segments` table, it will extract each of the features listed in features.txt.

## A.3 Aggregate Features

PhysioMiner `Aggregate` aggregates features for existing windows in the database. It accepts the following arguments:

```
 -a, --ami <arg>           Amazon Machine Image to launch instances
                           with (default is the public Physiominer AMI)
 -g, --aggregators <arg>   Comma-separated list of aggregators to use
                           (possibilities: mean,skew,trend,kurtosis,
                                           standard_deviation)
 -f, --features <arg>      Comma-separated list of feature names to
                              aggregate on
 -l, --lead                The lead to use in seconds
```

```
 -m, --lag                    The lag to use in seconds
 -n, --num-instances <arg>    Number of instances to create
 -p, --iamprofile <arg>       Name of IamProfile to use for EC2 instances
                              (must have access to DynamoDB, S3, and SQS).
 -t, --table <arg>            Name of window table with windows to aggregate on
 -y, --instance-type <arg>    Amazon EC2 Instance type to launch
                              (possibilities: r3.large, r3.xlarge,
                                     r3.2xlarge, r3.4xlarge, r3.8xlarge,
                                     c3.large, c3.xlarge, c3.2xlarge,
                                     c3.4xlarge, c3.8xlarge)
```

To aggregate features for existing windows, the command might look like:

```
java -jar aggregate.jar
   --features length,max,min --num-instances 3
   --aggregators mean,skew,trend --lead 600 --lag 1800
   --iamprofile myiamprofile --table ecg_windows
   --instance-type r3.large
```

Running the above line will create 3 `r3.large` instances using the public PhysioMiner AMI. For each window in the `ecg_windows`, it will calculate the predictor window using a lead of 600 seconds and a lag of 1800 seconds. It will use three aggregation functions (mean, skew, and trend) on three different beat features (length, max, min). These results will be stored in the table `ecg_windows_lead_600_lag_1800`.

# Bibliography

[1] Alivecor hardware. http://www.alivecor.com/what-is-it. Accessed: 2014-05-13.

[2] Amazon rds now supports 3tb and 30,000 provisioned iops per database instance. http://aws.amazon.com/about-aws/whats-new/2013/03/13/amazon-rds-3tb-30k-iops/. Accessed: 2014-02-11.

[3] Amazon s3 console. https://console.aws.amazon.com/s3/. Accessed: 2014-02-10.

[4] Athena health - about us. http://www.athenahealth.com/our-company/about-us/medical-practice-management.php. Accessed: 2014-05-05.

[5] Cardiac designs. http://cardiacdesigns.com/. Accessed: 2014-05-13.

[6] Delphi. http://delphi.csail.mit.edu/. Accessed: 2014-05-17.

[7] Dynamodb faq. https://aws.amazon.com/dynamodb/faqs/. Accessed: 2014-02-10.

[8] edf2mit. http://www.physionet.org/physiotools/wag/edf2mi-1.htm. Accessed: 2014-03-03.

[9] Heart disease facts. http://www.cdc.gov/heartdisease/facts.htm. Accessed: 2014-05-08.

[10] Launching instances with user data. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/user-data.html. Accessed: 2014-03-10.

[11] Make fitness a lifestyle with flex. http://www.fitbit.com/flex. Accessed: 2014-05-13.

[12] Physionet atm. http://www.physionet.org/cgi-bin/atm/ATM. Accessed: 2014-02-15.

[13] Pywavelets - discrete wavelet transform in python. http://www.pybytes.com/pywavelets/. Accessed: 2014-04-10.

[14] Qardio ios ecg, bp cuff for easy monitoring of heart health. http://www.medgadget.com/2013/06/

qardio-ios-ecg-bp-cuff-for-easy-monitoring-of-heart-health.html. Accessed: 2014-05-18.

[15] Qrs complex. http://www.studyblue.com/notes/note/n/nur-445/deck/5619821.

[16] Welcome to mlpys documentation! http://mlpy.sourceforge.net/docs/3.5/. Accessed: 2014-04-10.

[17] Wfdb applications guide. http://www.physionet.org/physiotools/wag/. Accessed: 2014-02-25.

[18] Una-May OReilly Franck Dernoncourt, Kalyan Veeramachaneni. beatdb: A large scale waveform feature repository. http://groups.csail.mit.edu/EVO-DesignOpt/groupWebSite/uploads/Site/nips2013beatdb.pdf. Accessed: 2014-01-05.

[19] M.L. Simoons and P. G. Hugenholtz. Gradual changes of ecg waveform during and after exercise in normal subjects. *Circulation*, 52.4, 1975.

[20] SINTEF. Big data, for better or worse: 90% of world's data generated over last two years, May 2013.

[21] Qibin Zhao and Liqing Zhang. Ecg feature extraction and classification using wavelet transform and support vector machines. *Neural Networks and Brain, 2005. International Conference*, 2, 2005.