BeatDB v3: A Framework for the Creation of Predictive Datasets from Physiological Signals

by

Steven Anthony Rivera

S.B., Massachusetts Institute of Technology (2016)

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

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2017

Author .....
Department of Electrical Engineering
and Computer Science
May 26, 2017

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

Certified by .....
Erik Hemberg
Research Scientist
Thesis Supervisor

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

BeatDB v3: A Framework for the Creation of Predictive Datasets from Physiological Signals

by

Steven Anthony Rivera

Submitted to the Department of Electrical Engineering and Computer Science
on May 26, 2017, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

BeatDB is a framework for fast processing and analysis of physiological data, such as arterial blood pressure (ABP) or electrocardiograms (ECG). BeatDB takes such data as input and processes it for machine learning analytics in multiple stages. It offers both beat and onset detection, feature extraction for beats and groups of beats over one or more signal channels and over the time domain, and an extraction step focused on finding condition windows and aggregate features within them.

BeatDB has gone through multiple iterations, with its initial version running as a collection of single-use MATLAB and Python scripts run on VM instances in OpenStack and its second version (known as PhysioMiner) acting as a cohesive and modular cloud system on Amazon Web Services in Java. The goal of this project is primarily to modify BeatDB to support multi-channel waveform data like EEG and accelerometer data and to make the project more flexible to modification by researchers. Major software development tasks included rewriting condition detection to find windows in valid beat groups only, refactoring and writing new code to extract features and prepare training data for multi-channel signals, and fully redesigning and reimplementing BeatDB within Python, focusing on optimization and simplicity based on probable use cases of BeatDB. BeatDB v3 has become more accurate in the datasets it generates, usable for both developer and non-developer users, and efficient in both performance and design than previous iterations, achieving an average AUROC increase of over 4% when comparing specific iterations.

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

Thesis Supervisor: Erik Hemberg
Title: Research Scientist

Acknowledgments

I could not have completed my thesis or work on BeatDB without the help of numerous individuals and groups along the way.

I'd like to thank Una-May O'Reilly for her kindness and direction throughout the development of this project. She made this process very bearable, meaningful, and fun, and I am very grateful to have a mentor as intelligent, insightful, and personable as her.

I'd also like to thank Erik Hemberg for preparing me for this thesis over Summer 2016 by coaching me through further development of the MLBlocks platform. He also provided very valuable advice and direction for the redesign of BeatDB and without him, I'm not sure that BeatDB would have come out as optimized and elegant as it did.

I'd like to thank Alejandro Baldominos for helping me understand how PhysioMiner worked and providing a groundwork for BeatDB v3 by developing the preprocessing module. Without Alejandro's help, I would have taken significantly more time to understand how this project worked and would not have been able to accomplish as much as I ended up doing.

I'd like to thank my mother, Claudia Rivera, grandmother, Maria Rosales, and aunt, Cecilia Rosales, for their constant support throughout my academic career and the sacrifices they made to ensure that I had these amazing opportunities available to me.

I'd like to thank Gabriela Carrillo for always being there for me every step of the way and believing in me even when I didn't believe in myself.

I'd like to thank Juan Huertas, Jordan Smith, and Reymundo Cano for giving me plenty of opportunities to destress and have fun when work became overwhelming.

I'd like to thank my band, Love and a Sandwich, for allowing me to continue pursuing my passion for drumming throughout my undergrad and graduate years of study.

Lastly, I'd like to thank my fraternity, Theta Delta Chi, for providing me with a nurturing, supportive, and academic environment over the entirety of my education at MIT.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

BeatDB is a project developed and maintained by the Anyscale Learning for All group (ALFA) in the Computer Science and Artificial Intelligence Laboratory (CSAIL) at MIT. It is a framework that allows users to input both single and multi-channel physiological data and analyze patterns and correlations within the data. It does this by parsing waveforms into individual chunks (such as beats, if the waveform can be expressed in them) while extracting additional features from each chunk in the population/extraction stage. Afterward, BeatDB uses this information, along with aggregated features over groups of chunks, to form a machine learning dataset in the condition detection/aggregation stage.

Normally when researchers want to conduct an experiment using physiological data, they need to spend time figuring out how to parse the data from the raw waveform. Given the variability in the way that data can be stored, this process tends to be specific to the type of data that the researchers have. Once the data has been parsed into a more readable and malleable form, the researchers need to develop a pipeline to process the data, usually making the pipeline very specific to the format of the parsed data. While this tends to lead to output that is very close to what the researchers want, the software they develop for these experiments is not typically reusable since it is so specific to the data and the way that the researchers processed it along the pipeline. Given the complexity of this process, it can take some amount of time and

use of resources to develop.

Why should researchers use BeatDB? BeatDB allows researchers and scientists to cut down on time needed for prediction studies and data processing without sacrificing any of the parameterization and specificity to the data possible with custom (and often single-use and unrefined) scripts. With BeatDB v3, users are given freedom to define their own features, validity functions, and input signal types (such as ambulatory blood pressure (ABP)) within the framework. They are also able to easily integrate any cloud based service within the framework, with Amazon Web Services (AWS) and local modes supported by default. Additionally, BeatDB v3 improves upon many issues and design concepts found in previous iterations of BeatDB, as seen in the comparison table 5.3 and expanded upon in section 5.2.

1.1 Development Background

BeatDB has gone through multiple iterations, with its key ideas having been previously implemented in three different languages and for two different instance frameworks (including a local mode, limited to a single worker and smaller test datasets). BeatDB v0, mentioned in detail in Waldin's thesis, was not a full system but a collection of single-use MATLAB and Python scripts used for both beat onset detection and the extraction of additional features from every detected beat. This iteration served as a prototype for the population/extraction stage of the next iteration of BeatDB.

BeatDB v1, which was a complete attempt at creating a distributed system for beat detection and feature extraction, condition detection, and feature aggregation, was the first attempt at making BeatDB a fully fledged application. Computation was run on a network of OpenStack instances at CSAIL, but as this was hard-coded into the framework, it prevented BeatDB from being accessible to many audiences. Additionally, BeatDB v1's design focus was on functionality, not general usability, rendering customization of the platform difficult. This iteration of BeatDB did not provide

features that would be expected of a user-based system and was difficult to adapt for use cases other than the ABP use case demonstrated in Dernoncourt's thesis on BeatDB.[7]

PhysioMiner, known as BeatDB v2, reworked the scripts that comprised BeatDB v1 into a modular and cohesive cloud based system hosted on Amazon Web Services (AWS). The algorithms in the scripts were rewritten in Java, with Python used for user-defined feature, condition, and aggregation functions.[10] At the time of development, AWS and the interest in cloud technologies was relatively new. Given this, and the need for a user-based version of BeatDB, ALFA implemented PhysioMiner on top of AWS. The scripts for the "analytics sub-framework" were the direct inspiration for PhysioMiner, with the four steps outlined in the hypothesis testing framework section of the NIPS workshop paper[8] becoming the four major stages of the PhysioMiner system.[8]

Since its development in 2014, PhysioMiner remained in use in ALFA group until December 2016 after a large processing job unveiled multiple issues with the framework and its scalability potential, leading to numerous message timeouts, delayed processing time for each message, and a significant amount of money being spent on the computation (one specific computation over the entire MIMIC-II ABP dataset cost over $1500 dollars). Although PhysioMiner was built with the BeatDB v1 design in mind, its design was too over-engineered and intertwined with AWS, limiting its effectiveness for large tasks. Feature creep and poor design choices caused PhysioMiner to bloat over years of maintenance, making it more and more difficult for new researchers to use the platform. Issues with the AWS framework, primarily those caused by message timeouts in the queues, drastically affected computation time and cost. The issues stemming from this large computation forced ALFA to reevaluate PhysioMiner and compare its implementation with the BeatDB v1 design.

After evaluation, multiple objectives were created for a new iteration of BeatDB, focusing on usability, efficiency, and correctness, along with the implementation of new

features that allow for more complex generation of feature data from beats. This iteration of BeatDB, called BeatDB v3, and the design and implementation of this new iteration, are the focus of this thesis.

1.2 Objectives

There are a number of research questions for this body of work, most of them related to the creation of BeatDB v3. These are listed and discussed below.

Does the PhysioMiner system achieve comparable correctness to the BeatDB v1 system?

Given the differences in system design and implementation, the best way to compare PhysioMiner to BeatDB v1 is to compare the results of specific computations on both systems. The process of verifying the correctness of the PhysioMiner software will allow us to determine how similar results of learning on generated datasets from PhysioMiner's implementation are to BeatDB v1's design, and as a result, will guide our decisions regarding the design and implementation of BeatDB v3.

How can the BeatDB system be more usable?

Though each subsequent iteration of BeatDB has attempted to make the software easier to use for non-developers, there is still more progress to be made in simplifying the method of control over the parameters that the user has. The process of running the PhysioMiner software, both locally and on the cloud, can be confusing, even for advanced users. New iterations of BeatDB should be designed with a focus on the technical ability of expected users, but not to the point that functionality and efficiency are hindered. Additionally, the source code of the software should be clear and understandable for developers who wish to modify the source code so that the system stays efficient over time and the inevitable maintenance it will be put through.

How can the BeatDB system be more efficient, in terms of performance, software design, and implementation?

PhysioMiner became bloated over multiple years of use and modification and, as a result, started to deviate from its original design. After performing a code analysis of PhysioMiner, we discovered multiple ingrained issues with the implementation that have a large negative effect on computation run-time. When processing data, PhysioMiner does not strive for efficiency, requiring multiple full passes over file data for a single population task. Additionally, the process for importing feature functions is inefficient, forcing PhysioMiner to wait for file I/O for every feature calculation and requiring that each feature script be redownloaded from S3 for every function call related to that feature.

Given the complexity of the PhysioMiner system and how rigid various parts of the design are, attempting to fix PhysioMiner's low-level implementation issues would require many resources and would ultimately lead to a weaker system, especially when considering the amount of patching that would need to be done to add functionality and increase efficiency in the software. Rather than fix PhysioMiner and add more bloat to the source code, ALFA group decided that rewriting BeatDB in Python will allow for the simplification of various parts of the system design.

Coding in Python makes it easier to add new software features to the system, and having the hindsight of what issues arose in BeatDB v2 allows for revision of system design mistakes made in the past. Performance-wise, coding in Python will heavily reduce the number of subprocess calls made for the user-defined scripts necessary for certain steps. Additionally, coding in Python will greatly speed up the system as processes will no longer need to wait for the costly file I/O overhead for each feature function's results.

Does the BeatDB v3 system achieve comparable correctness to the PhysioMiner and BeatDB v1 systems?

Because BeatDB is being reimplemented based on previous designs, it needs to be reevaluated for correctness and consistency with previous iterations to ensure that similar results are generated at each newly developed stage and that the new BeatDB software is performing as expected. Executing correctness tests, similar to those used to compare PhysioMiner and BeatDB v1 at the beginning of this work, will give confidence in the consistency of BeatDB v3's results with those from previous systems.

How can functionality be added to BeatDB v3 to allow for more customization, input filtering, and the ability to process multi-dimensional waveforms?

Though this thesis has been focused on the creation of the BeatDB v3 system, its initial objective was to add support for multi-channel features and multi-dimensional waveforms to PhysioMiner. Discovery of the issues with PhysioMiner led to a focus on a new iteration of BeatDB, but multi-dimensional waveform support remains a priority in the development of BeatDB v3. The design of BeatDB v3 made it simple to adapt the codebase for multi-dimensional waveforms and multi-channel features, allowing users to define their own multi-dimensional signal types and multi-channel features without much more complexity than is already required for customized single-channel signal types and features.

BeatDB v3 also includes standalone features related to the filtering of input file sizes and preprocessing of input data to allow the user to have greater control over their computations. Following its design goals, BeatDB v3 is more user-friendly, only requiring a configuration file to parse all arguments for all steps at once in order to simplify the execution of the system. Lastly, users will be able to add support for cloud frameworks of their choice, allowing BeatDB to be run on frameworks other than AWS.

1.3 Organization

- Chapter 2 presents the architecture of the BeatDB design and its realization in previous iterations of the system
- Chapter 3 discusses PhysioMiner in detail, focusing on output comparisons with findings from Waldin's thesis and modifications made to the system
- Chapter 4 provides motivation for and details the design of BeatDB v3
- Chapter 5 details the testing and demonstration of BeatDB v3 with original ABP data used in previous iterations of BeatDB
- Chapter 6 concludes this thesis and discusses future work and goals for the BeatDB v3 platform
- Appendix A contains technical information and specifics regarding the implementation and usage of BeatDB v3
- Appendix B contains pseudocode for specific algorithms that are critical to BeatDB v3

Chapter 2

Background

2.1 Related Work

Despite research into similar frameworks, it seems that BeatDB is one of the few frameworks that combines several of the specific concepts behind it, such as onset detection, feature aggregation, and the ease of customization coupled with its user-based design. BeatDB is a novel system primarily because of its focus on flexibility toward multiple signal types and the combination of its provided features, such as the coupling of beat detection along with condition detection and window aggregation. Despite this, there is a large amount of work related to different parts of BeatDB, as feature extraction of signal data and prediction of medical events are large fields of interest in computing, among other components of BeatDB.

2.1.1 Feature Engineering

Feature engineering is a popular subject for research because features impact model prediction accuracy, meaning that findings can have a large impact on future machine learning research.[19] Because many of the signal types are waveforms, using wavelets, particularly wavelet transforms, is a common approach for feature engineering and extraction research. In fact, wavelets have been shown to outperform the fast Fourier transform as a spectral analysis tool for detecting brain diseases.[1] A paper

from 1997 describes the use of artificial neural networks along with the wavelet transform to classify EEG signals between normal, schizophrenic, and obsessive-compulsive classes.[12] One group of researchers uses the stationary wavelet transform on EEG data to"reduce artifacts from scalp EEG recordings to facilitate seizure diagnosis/detection for epilepsy patients."[5] In the hardware realm, researchers have designed a wavelet-based ECG detector for use in implantable pacemakers, noting that "wavelet-based detection algorithms [are] generally considered as one of the most effective algorithms."[19]

Despite this, there is still value in researching other features. An algorithm proposed in 2013 uses local extreme values and their dependencies to extract locations of ECG deflections, allowing the algorithm to form the beat and detect anomalies in real-time.[29] Another group of researchers proposed a method for monitoring cerebral autoregulation in patients that focuses on an improvement to the signal abnormality index algorithm described in the ABP section above. By adding two simple summation features, the researchers were able to greatly improve the specificity of the signal abnormality index for ABP data.[35]

Feature extraction can also be automated via usage of deep belief networks, as shown in the following examples. A group of researchers used this approach to predict emotions by "automatically [extracting] features from raw physiological data of 4 channels in an unsupervised fashion and then [building] 3 classifiers to predict the levels of arousal, valance, and liking based on the learned features."[32] This implementation of deep belief networks is unique because it is the first attempt at using them to predict emotions, but deep belief networks have been used frequently with physiological data in the past, showing utility in both handwriting recognition[15] and classifying between stages in a sleep cycle[14], among other applications.

2.1.2 Prediction of Medical Events

The prediction of medical events is a popular field of research as it has various applications in the real world. A medical event is one of interest that usually denotes some sort of issue with a patient, such as an acute hypotensive episode or hemodynamic instability. Researchers at MIT used symbolic analysis and clustering of cardiovascular signals, available via ECG data, to predict "unexpected events" of interest.[28] The researchers used morphological features to partition beats into classes that could be represented by symbolic strings "corresponding to the sequence of labels assigned to the underlying unit." Afterward, they searched "for significant patterns in the reduced representation resulting from symbolization," allowing them to quickly analyze the data for irregularity with the idea that "[symbol] variations that are unlikely to occur purely by chance" are likely to be the most medically relevant.[28] The researchers had success with their techniques, uncovering multiple examples of heartbeat irregularities while achieving very high consistency with cardiologist opinion.

Additional research has been conducted to attempt to predict more specific cardiovascular events. Two researchers at the Bhoj Reddy Engineering College for Women have developed an algorithm for detecting arrhythmia from ECG signals using ST segment and QRS detection. Once these regions have been detected, signal extraction occurs by using a combination of discrete wavelet transformation and support vector machine techniques to uncover ECG features and classify each beat as arrhythmic or normal.[23] Similarly, research conducted at the Khulna University of Engineering & Technology also used support vector machines to detect cardiac diseases such as different types of arrhythmia and myocardial infarction (heart attacks).[30] Researchers at Leiden University Medical Center have analyzed the spatial QRS-T angle (SA) as a measure of ECG concordance and determined that patients with emerging heart failure after a heart attack are more likely to have ECG waveforms that are less concordant.[6]

Research related to prediction is not just focused on single signal types. Another

group of researchers from National Taiwan University proposed a multi-modal analysis of physiological signals to determine patient functional outcomes after suffering from a stroke.[13] This finding is particularly notable as a joint analysis of the ECG, ABP, and PPG signal types outperforms single-modal frameworks using only one of the signal types while yielding performance comparable to the current diagnosis scale for stroke victims known as the National Institutes of Health Stroke Scale (NIHSS).

It is also worth noting that research related to the prediction of medical events is not restricted to software methods. A group of doctors developed a device intended for use in emergency rooms that can detect if a patient has a traumatic brain injury or brain bleeding. The device, called the AHEAD 300, works by detecting EEG waves from a reclined patient for up to 10 minutes. Afterward, the device extracts multiple features and analyzes them to determine if the electrical activity in the brain is normal or delayed or if the sides of the brain are coordinated or out of sync. Given the portability and accuracy of the device, along with its ability to detect early head injuries, this research shows promise for multiple fields and uses.[11]

2.1.3 Computing Frameworks

Similar to PhysioMiner, researchers at Universiti Technology Malaysia developed a framework for cloud computing with ECG big data, motivated by the lack of big data computing in physiological analysis. By using Hadoop along with MapReduce to parallelize ECG analysis, the researchers hope their framework encourages others to pursue physiological big data ventures. Their work shows a speedup of almost 30x using MapReduce with 5 nodes and a speedup of 7x using MapReduce with 1 node compared to not using MapReduce when analyzing ECG.[34]

Given the complexity of EEG signals, three researchers in China have developed a framework that efficiently utilizes hybrid feature extraction, involving autoregressive models, wavelet transforms, and sample entropy to generate complex and descriptive features, and feature selection methods to reduce dimensionality and redundancy of

the feature data. By combining these methods, the researchers are able to compute a large number of features and consequently remove redundancies and nondescriptive features that arise, resulting in a set of features that accurately describes the data.[21]

2.2 Medical Data

The data used throughout the paper primarily comes from the *Multiparameter Intelligent Monitoring in Intensive Care II* (MIMIC II) waveform database. MIMIC II is a "freely available database... intended to support epidemiologic research in critical care medicine"[22] and is currently in it's third version.[16] It consists of two major databases: the clinical database, which contains comprehensive and de-identified[9] clinical information "from bedside workstations as well as hospital archives" about Intensive Care Unit (ICU) patients, and the waveform database, which stores "continuous high-resolution physiologic waveforms and minute-by-minute numeric time series (trends) of physiologic measurements."[1] The waveform database, used for its inclusion of physiologic waveforms, consists of 3 TB of data and contains 23,180 patient records, 17,468 of which come from adult patients.

A record is analogous to an office visit, as multiple records could belong to one patient at different times, though there are accidental instances of a record containing multiple patients separated by a gap containing no signals. Each record contains some subset of the following physiological waveforms.[2]

- A set of ECG (electrocardiographic) waveforms
- BP (continous blood pressure) waveforms include:
  - ABP: arterial blood pressure (invasive, from one of the radial arteries)
  - ART: arterial blood pressure (invasive, from the other radial artery)
  - CPP: cerebral perfusion pressure
  - CVP: central venous pressure

[1] https://physionet.org/mimic2/
[2] https://physionet.org/mimic2/mimic2_waveform_overview.shtml

  - FAP: femoral artery pressure
  - ICP: intracranial pressure
  - LAP: left atrial pressure
  - PAP: pulmonary arterial pressure
  - RAP: right atrial pressure
  - UAP: uterine arterial pressure
  - UVP: uterine venous pressure

- PLETH: uncalibrated raw output of fingertip plethysmograph

- RESP: uncalibrated respiration waveform, estimated from thoracic impedance

2.2.1 Issues with Waveform Data

PhysioNet describes multiple issues that exist throughout the data in the waveform database, e.g. the database not containing both waveform and numerics records for 10.75% of all records in the database and some waveform signals not being available for the entire duration of a record. Only one of the issues directly affects our research, with other issues related to signal types not yet implemented within BeatDB v3.[3]

Gaps and patient identification

As mentioned above, some records may contain information from multiple patients separated by a gap containing no signals. This is related to the method of extraction used to gather the data for each waveform. Each waveform and numerics record comes from raw data dumps that were collected from bedside monitors in intensive care units. In some cases, monitors were not reset between patients, or monitors were disconnected from patients for a non-trivial amount of time, causing gaps in the waveform. Since the raw data did not contain patient identifiers, it is unclear which gaps are indicative of a new patient being connected to the monitor or the same patient being reconnected to the monitor. An attempt to mitigate this issue

[3] https://physionet.org/physiobank/database/mimic2wdb/#technical-limitations

was made in MIMIC-II v3, which split raw data files with gaps of an hour or more into separate records, though this may also have split patients in the same visit into multiple records given the large amount of files that were split (48.4% of all files contained gaps of an hour or more).[4]

2.2.2 Using the Data

Given that the focus of this work has been the redesign of the BeatDB system, most of the data from previous builds of BeatDB was used for testing throughout the development and use of BeatDB v3. Since the ABP data has been used in more iterations of BeatDB and is more understood in our documentation, the preprocessed ABP data from Waldin's thesis[31] was primarily used to compare PhysioMiner and BeatDB v3 for consistency in results.

Arterial Blood Pressure (ABP)

There is a significant amount of ABP data available in MIMIC-II, amounting to over 240,000 hours and 2 TB of uncompressed waveform data in the form of 108 billion samples.[7] Despite this, only 6,232 records, stored as individual .edf files, have ABP signals (26.9% of the waveform database), though not every ABP signal has enough beats to be useful for the type of analysis that BeatDB provides. Following the procedures from previous iterations of BeatDB, each record's ABP waveform data is parsed into beats using the onset detection algorithm[36] available in the WFDB platform.[20] Each beat is marked as valid or invalid depending on its *signal abnormality index*.[25] The signal abnormality index is a set of constraints that all valid beats must satisfy. The constraints used in this project are listed below.

- Systolic blood pressure must be less than or equal to 300 mmHg
- Diastolic blood pressure must be greater than or equal to 20 mmHg
- Mean arterial pressure must be between 30 and 200 mmHg, inclusive

[4] https://physionet.org/physiobank/database/mimic2wdb/#technical-limitations

- Heart rate must be between 20 and 200 bpm, inclusive
- Pulse pressure must be greater than or equal to 20 mmHg
- Number of samples in a beat must be greater than or equal to 10
- The difference between beat i and beat $i-1$'s systolic and diastolic blood pressures must both be less than or equal to 20 mmHg
- The difference between beat i and beat $i-1$'s duration (in seconds) must be less than or equal to $\frac{2}{3}$ of a second

Additionally, we added requirements for the number of samples the beat contains, with a user-defined upper bound determining if a beat is considered a *gap beat*, which is a special type of invalid beat that flags a gap or abnormality in the signal at this beat. Given these constraints, 77.2% of the usable ABP-containing records had at least 80% valid beats and 60.5% had at least 90% valid beats.[7] Given this, along with the difficulty of recording ABP without noise both physically[18] and digitally,[17] we hope that the size of the dataset and the attempts at detecting and removing noisy beats from consideration using the signal abnormality index overcome the inherent noisiness of the data.

Electroencephalogram (EEG)

EEG signal data was used in the development and creation of BeatDB v3 to test multi-channel and multi-sample feature support within the platform. The data used for testing comes from data used to determine the effectiveness of artifact removal methods in motion data.[26][27] As a result, the EEG data contains two EEG waveforms and two sets of three accelerometer signals, with each representing a different axis in physical space. The EEG data can be found in MIT format (a data format consisting of separate header (.hea) and data (.dat) files) coupled with .trigger files for each data file on the PhysioNet website.[5] MIT format can be converted to .edf format, which is formally used as input by BeatDB v3, but because the data contains

[5] https://physionet.org/physiobank/database/motion-artifact/

multiple signals, the .trigger files are required to properly align the signals. The motion artifact data was ideal for testing multi-channel and multi-sample features, as the data in each channel could be aggregated over a rolling window and also allows for simple multi-channel features to be generated easily given the layout of the data. More information about this can be found in section 4.2.3.

Given the design issues with parsing EEG, as the signal type does not contain beats and condition detection with EEG chunks is not fully understood, BeatDB v3 does not formally support EEG data. Beatless data has not been tested for correctness with the platform and is inherently different from waveforms with beats, such as ABP or ECG. However, BeatDB v3 has been designed in such a way to allow for simple implementation of EEG (as was done for the development of multi-channel and multi-sample feature capability), allowing future researchers to fully understand how EEG and other beatless signals can be interpreted by BeatDB.

Chapter 3

BeatDB

BeatDB was first designed in 2012, and though it has been reimplemented fully on at least three separate occasions, the basic ideas and concepts behind it have remained constant. This section will discuss a high-level view of BeatDB and attempt to outline the theoretical system behind it without giving attention to implementation specifics. Afterward, this section will go into detail about the previous implementations of BeatDB and how they have contrasted with one another. Lastly, modifications to PhysioMiner (the second iteration of a fully fledged BeatDB system) made in Fall 2016 in an attempt to better align PhysioMiner with previous iterations of BeatDB will be discussed.

3.1 BeatDB Design

Before discussing how the most abstract BeatDB system would be split, insight should be given on what is normally required when generating a machine learning dataset. To start, researchers gather raw data from some sort of database. A popular one for physiological data is MIMIC-II (see 2.2 for more details). Once the raw data has been gathered, the researchers need to write code to convert the data into a program-useful format. Because raw data from different sets of physiological data almost always vary in shape and composition, these scripts tend to be designed for single-use. Once the data has been parsed, additional features may be extracted from

each row in the resultant chunks, or groups of sample values. Once this has been done, a lag/lead rolling window algorithm is performed over the resultant chunks in order to generate the predictive machine learning dataset. Since the amount of work required to prepare data for machine learning is non-trivial, researchers could benefit from having a software system that automates this process for them in a distributed way.

To adapt the problem to BeatDB, a clear division in the main dataset generation algorithm needs to be found. In theory, BeatDB can most simply be split into two major modules. Overall, its goal is to parse any physiological data into chunks of some sort, such as beats, and then use this information to form a dataset that can be analyzed with machine learning techniques. As such, BeatDB can be thought of as a simple system that focuses on flexibility of input and ease of use by running its highly generalized modules sequentially, with the data from the first module being used in the second module to form the final output of the system in a pipeline fashion.

3.1.1 Beat Object Generation

In the first module of BeatDB, the physiological data is parsed into beats or chunks, depending on the nature of the signal type. If the signal type can be parsed into beats that can be detected using a beat onset detection algorithm, then it is trivial to split the raw waveform into smaller chunks (just split the waveform into the beats themselves). Some signal types, such as EEG or accelerometer data, do not have groups of signals that can be parsed into beats. Instead, researchers define their own metrics for splitting the raw waveform, such as splitting the raw waveform into evenly sized chunks, making a new chunk after noticing some signal value, or determining where chunks lie by detecting some sort of trend. Regardless of how this is done, BeatDB should be flexible enough to adapt to this requirement.

During this process, BeatDB should also be able to extract features from the chunks it identifies in the raw waveform. For maximum flexibility, BeatDB should allow

for custom features to be imported and it should allow the user to select between existing features to compute using each the sample values of each chunk. Given these requirements, the end result of the Beat Object Generation module should be a set of chunks, representing the original dataset exactly, each with some set of customized computed feature values based on the sample values contained within the chunk.

3.1.2 Dataset Preparation

In the second module of BeatDB, the chunks generated from the previous module are used to create a prediction dataset for machine learning analysis. As with the previous module, this module has two goals. First, the module needs to determine which consecutive groups of the chunks are representative of some sort of detectable condition based on arguments given from the user. The user determines the condition to detect and defines the procedure for determining the condition. Additionally, the user gives values for *lead*, *lag*, and *condition_window_length*. These values outline the length of a rolling window that will be applied over the raw waveform (comprised of consecutive chunks at this point). This rolling window consists of three smaller windows: the predictor window with *lag* duration, the lead window with *lead* duration, and the condition window with *condition_window_length* duration, all with durations defined in terms of seconds. The chunks in the condition window are given to the condition detecting procedure to determine if that group is representative of the condition or not. A visual representation of this rolling window, referred to as the group window, is shown in Figure 3-1.

The second goal of this module requires that aggregate features are computed using a prediction window that occurs some time before the condition window in order to ensure that we have adequate prediction data related to the condition window. These aggregate features are computed from the features of the chunks found in the lag (predictor) window. The combination of these goals allows this module to create a machine learning dataset, with both classifications from condition windows and aggregation features from predictor (lag) window subwindows present in each group

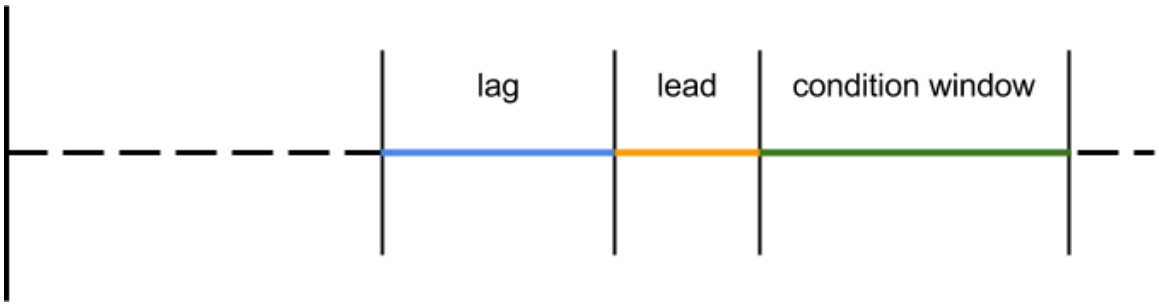


Figure 3-1: This figure shows a group window imposed over a waveform, with the group window consisting of a lag (predictor) window, lead window, and condition window. This is the structure of the rolling window that is first applied to the very beginning of the waveform and rolled over to the end of the waveform.

window recorded by the module. At this point, the user can use this dataset to perform machine learning analysis on the results and attempt to find some sort of meaningful correlation between the prediction data and the detected conditions.

3.2 Previous Implementations

As mentioned, BeatDB has been realized in multiple implementations in the past few years. In this section, details regarding these past implementations and their specifics and shortcomings will be discussed.

3.2.1 BeatDB v0

Though this iteration of BeatDB was not referred to as BeatDB, it essentially outlines an implementation of the first module described above and served as a critical building block to the development of the BeatDB system. The Beat Feature Database described in section six of Waldin's Master's thesis[31] was developed to parse ABP data into beats with features. Waldin stored the raw waveforms of each of the segments found in MIMIC-II and then, after preprocessing the data to remove noise, performed onset detection on them to find each beat. During this process, Waldin marked beats as invalid or valid but also checked if beats were abnormally long in duration (greater than 6 seconds) and marked them as *gaps*. In his words, "a gap

indicates a disturbance in the signal that is significant enough that the data prior and posterior to the gap should not be considered as belonging to the same contiguous segment."[31] In addition to finding the validity of each beat, Waldin computed various beat features for each beat, completely fulfilling all requirements of the Beat Object Generation module described in section 3.1. A very simple diagram of the system overview of BeatDB v0 can be seen in figure 3-2.

Even though this iteration did not include any attempts at detecting conditions within groups of beats or aggregating features and was not designed with flexibility to input in mind, it laid the groundwork for future iterations of BeatDB by introducing the idea of a beat database that takes physiological data and outputs beats with additional features. Waldin used this database for further analysis of blood pressure data, using the data in a variety of experiments that combined various classification methods and parameters to determine that more advanced features and classification methods must be used to predict blood pressure for a general patient.[31]

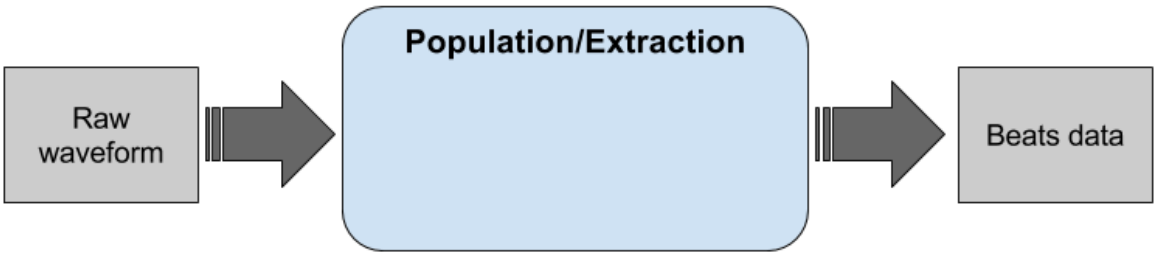


Figure 3-2: This figure shows a very simple diagram of the system overview of BeatDB v0. It only contains the population/extraction module, which takes in raw waveform ABP data and outputs Beats data, with each beat object containing extracted features.

3.2.2 BeatDB v1

The first true iteration of BeatDB was primarily developed by Franck Dernoncourt and will be referred to as BeatDB v1 throughout this thesis. A more fleshed out version of the Beat Feature Database that Waldin proposed, BeatDB v1's main objec-

tives were multi-level parameterization (primarily for flexibility of experimentation), lossless storage, and scalability.[7] It expanded upon the beat detection and feature extraction concepts presented in BeatDB v0 by adding a joint condition detection and feature aggregation module, introducing the rolling window condition detection algorithm used throughout future iterations of BeatDB (mentioned above in 3.1.2). For a diagram of the BeatDB v1 system overview, see figure 3-3.

Given the first main objective of BeatDB listed above, this module was developed with flexibility to parameters in mind, allowing the user to specify the event to detect, which beat features to compute, how to aggregate features, and the machine learning algorithms and evaluation metrics used to analyze the resultant dataset. Despite this, setting these parameters was not particularly user friendly, requiring users to have familiarity with the codebase to modify these arguments. As for its other main objectives, BeatDB v1 initially employed a master/worker architecture introduced by Waldin called DCAP (A Distributed Computation Architecture in Python)[1] and later moved to a multi-worker architecture in which workers were synchronized by a common result database in order to limit code overhead for connections between instances. To store lossless data, NFS servers were used so that workers could easily access data files in a shared way. Given the inexpensive nature of using OpenStack and NFS servers at CSAIL, BeatDB v1 was able to achieve scalability on large MIMIC-II ABP datasets.

Interestingly, BeatDB v1 had the most feature functionality of all iterations, expanding beyond condition detection and feature aggregation by directly computing a user-specified evaluation metric using a machine learning algorithm of the users choice. No future iteration has been concerned with providing this sort of analysis to users, focusing instead on the generation of a dataset that is immediately ready for machine learning analysis. Additionally, BeatDB v1 incorporated alternative complex methods of condition detection, using wavelets and a Gaussian process for parameter optimization on the same ABP problem to show the effectiveness of their implementa-

[1] http://byterial.blogspot.com/2013/02/dcap-distributed-computation.html

tions within the system. It is likely that these methods were not propagated forward to future iterations of BeatDB because of their much lower average AUROC scores compared to the rolling window condition detection algorithm and the complexity of implementation that each method required.

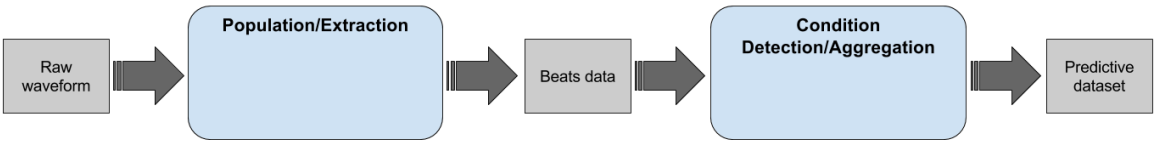


Figure 3-3: This figure shows a simple diagram of the two modules that comprise BeatDB v1. BeatDB v1 added onto BeatDB v0 by adding a condition detection/aggregation stage that outputs a predictive dataset of lag subwindows.

3.2.3 PhysioMiner

PhysioMiner is comprised of four major steps, outlined in detail throughout Gopal's Master's thesis and in documentation maintained by ALFA[3]. In each step, the PhysioMiner master instance creates or alters an existing DynamoDB table by adding messages to an SQS queue related to the step which are processed by worker instances who record and write results to the tables. Each step that occurs later in PhysioMiner is dependent on the tables generated by the steps before it. For a diagram and further explanation of the system architecture of PhysioMiner, refer to 3-5 at the end of this chapter.

What motivations were there for building PhysioMiner if a working version of BeatDB v1 existed? PhysioMiner is, more explicitly, the Amazon Web Services (AWS) system architecture used to run BeatDB, albeit a customized version of BeatDB that differs from v1 and is written in Java. As such, PhysioMiner is frequently referred to as an iteration of BeatDB rather than a framework that runs a BeatDB instance. PhysioMiner's intent was not to overtake BeatDB but to provide a way to run the software on a modular and new (at the time) cloud service. As development progressed, PhysioMiner became an iteration of BeatDB in its own right, given that BeatDB was rewritten entirely within PhysioMiner. In this sense, PhysioMiner's main motivation

was to expand BeatDB beyond the CSAIL servers and into a new cloud architecture with the hopes that the system could readily be used by a broader range of researchers than previously allowable with BeatDB v1.

Additionally, BeatDB v1 was not as user oriented as desired for a system meant to be used by numerous researchers on different projects. PhysioMiner allowed for users to customize each parameter as they called and initialized the script, giving the user more control over the system than BeatDB v1. Users were also allowed to create their own feature, aggregation, and filter functions in Python, which would be used by PhysioMiner for the stages it would run.

Rather than being broken up into two stages, as the conceptual BeatDB and BeatDB v1 did, PhysioMiner breaks BeatDB into four stages, consisting of population, feature extraction, condition detection, and feature aggregation. These steps can be seen as submodules of the two stages present in the other BeatDB iterations and are briefly discussed below.

Population

Population is the first step of PhysioMiner. It reads raw waveforms in the data folder of the root S3 bucket and populates a *segments* table. The *segments* table relates the raw waveform files to *segment_ids*. Population also runs over each waveform and performs an onset detection algorithm to distinguish between the individual beats. Finally, PhysioMiner runs through the beats, extracting features from them that are specific to the signal type of the data while assigning a validity to them based on certain conditions.

A beat has one of three possible values in its *valid* field: 0 for invalid, 1 for valid, and 2 for gap (in the original PhysioMiner implementation, valid was a boolean and had no way of distinguishing between gap beats without directly checking the duration. See 3.3 for more details.). When considering ABP data, beats must pass validity checks related to having systolic pressure, duration, and other feature values

between acceptable ranges (this refers to an implementation of the signal abnormality index mentioned in 2.2.2). If the beat has a duration of greater than 6 seconds, which is impossible for a human heartbeat and implies some sort of loss of signal or disconnection from the sensors, then the beat is assigned a validity of 2.[31] Once all beats have been analyzed for validity, they are written to a *beats* table in DynamoDB.

Feature Extraction

The next step is feature extraction, which iterates over a specific *beats* table and adds additional features, specified by the user, to each beat based on analysis of each waveform. Features to be added must be specified in the command that runs feature extraction. Each feature must have a corresponding Python script containing its logic placed in a subfolder for the data type (ABP, ECG, etc) inside of the $features/$ folder in the S3 bucket. When feature extraction is run, it will add values for the new features not yet present in each individual row in the *beats* table and overwrite values for features with the same name that have been recalculated (if the script has not changed, it will overwrite with the same information assuming the raw input data is the same).

This step is run in conjunction with population by default but can be rerun afterward to add more features to the individual beats in the specified *beats* table. Population and feature extraction can be seen as two submodules of the Beat Object Generation module in the conceptual view of BeatDB discussed in 3.1.

Condition Detection

The third step in PhysioMiner is condition detection. This step is described as it exists in the latest update to PhysioMiner, and differs from its original implementation (see 3.3). When a beat is checked for validity, it may be flagged as a gap beat, described in the population step above. If a beat is a gap beat, any beats after and including this gap beat may not be part of a contiguous segment as the duration of the gap beat indicates a loss of signal. In order to only consider valid groups of the patient's raw

waveform, PhysioMiner must preprocess the waveform and find where gap beats occur and where these gap sections end. A *valid beat group* is a section of the patient's raw waveform starting with a valid beat and containing no gap beats. This guarantees that valid beat groups are contiguous sections of the waveform and are not noise.

After generating the valid beat groups (VBGs), a sliding window algorithm is run over each of them. The sliding window is initially formed at the start of the VBG, with PhysioMiner calculating the sliding window's end beat by considering the duration of lag, lead, and the condition window. If the end beat is beyond the bounds of the VBG, the algorithm continues to the next VBG. If our sliding window fits in the VBG, the condition is determined from the condition window using a condition script. If the condition window matches the condition, the window is saved and the sliding window is moved forward by an entire sliding window length so that it starts at the end beat of the current sliding window. Otherwise, the sliding window is only moved ahead by *lag* seconds.

For example, let's assume a sample waveform has 800 seconds of data and no gaps and the lag and lead durations are both 10 s while condition window length is 60 s. Assume that the first group window (the first step of our rolling window algorithm) starts at the very first beat, and each beat is 1 second long. Detecting the condition in the condition window would make the next rolling window start at the 81st second (jumping over the condition window entirely), while not detecting the condition would make the next rolling window start at the 11th second (moving ahead by lag seconds). If gaps had existed in this waveform, the algorithm would simply check if a gap beat existed in the group window's range every time it had a new start index, and if there was a gap beat in its range, the algorithm would start the group window at the first valid beat after this gap beat. A simplified visual representation of this can be seen in figure 3-4.

Once the end of the last VBG is reached, a user-defined filter script may run on the discovered windows, allowing the user to only see a specific number of windows

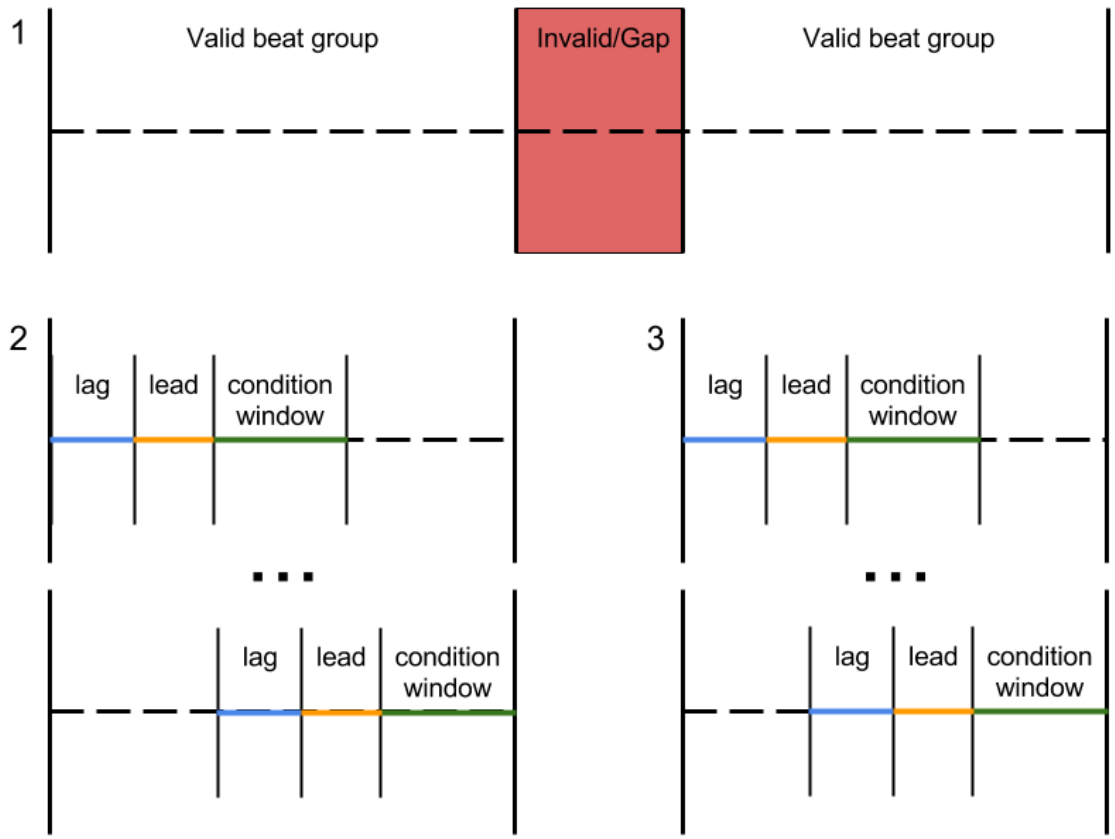


Figure 3-4: This image shows how VBGs are found and used in condition detection. 1 shows a waveform being split into valid beat groups and invalid/gap sections (a section that starts with a gap beat). 2 shows the first VBG being processed by condition detection, which starts by placing the sliding window at the start of the VBG and then running the sliding window algorithm on the VBG until the window reaches the end of the VBG. Afterward, the sliding window algorithm starts at the start of the next VBG and is run until it reaches the end of the VBG, as shown in 3. By only detecting conditions over VBG's, we ensure that the generated dataset contains data containing the least noise and possibly save on computation time by not needlessly generating data over invalid regions.

matching some sort of user-defined criteria. Finally, the resultant windows are written to the *windows* table (with no filter script, all windows are written). Each sliding window is represented by a row in the *windows* table, with each row containing information such as the condition classification, beat id's of the start and end beats, lead, lag, and condition window length. (In the original PhysioMiner implementation, each sliding window in the *windows* table did *not* contain lag or lead since each entry was only representative of the condition window. See 3.3.2 for more details.)

Feature Aggregation

The last step of BeatDB is feature aggregation. Feature aggregation reads from the *windows* table generated in the previous step with the aim of aggregating features over each predictor (lag) window. Users specify how many subwindows the lag window will be divided into and which user-defined aggregator functions to run over specified features from the extraction step. New, aggregate features for each subwindow from the lag window are generated, with each group window existing as a row in the *windows* table. The number of new features added to each row is found by multiplying the number of aggregator functions by the number of features, and the number of new rows is found by multiplying the number of rows in the windows column by the number of subwindows. For example, if the user picked two subwindows and supplied four aggregators and two features, the new dataset would have twice as many rows as the windows table and each row would have 8 more features (one aggregate feature is created for each feature and aggregator pair).

After generating the lag window aggregate features, a row with the old and new features for each subwindow is added to the $windows\_lead\_x\_lag\_y\_subwindows\_z$ table, where x is lead in seconds, y is lag in seconds, and z is number of subwindows. Since each window and subwindow still has the same classification as generated in condition detection for it's parent group window, this table is useful for learning, as one can use the aggregate features to try to predict the label. Using additional processing scripts, the user can generate a .csv file from the table and port it into any machine learning framework for analysis, keeping in mind the columns that need to be removed or ignored for processing, such as those containing id values and other string-based information.

3.3 Modifications to PhysioMiner

After completing the implementation of PhysioMiner, the system was tested using ECG data. However, PhysioMiner did not adequately test the accuracy of generated

datasets when using the ABP data from the BeatDB v1 experiments. The first major task of this thesis work required searching for the NFS server used with BeatDB v1 for code or experiment results from Dernoncourt's work. In this process, inconsistencies between BeatDB v1 and PhysioMiner were found and investigated, leading to further implementation on the PhysioMiner platform. This section details the comparison between PhysioMiner and BeatDB v1, leading to the discovery of the cause of the inconsistencies, and a high-level overview of the process of modifying PhysioMiner, giving an analysis of its results.

3.3.1 Comparison with BeatDB v1

Despite the searches through the lab's NFS server, code for onset detection, the wavelet feature implementation, and other internal processes were discovered, but a complete working copy of BeatDB v1 was not found. However, a significant amount of resultant aggregated datasets created by BeatDB v1 were found, allowing for comparison between the output of PhysioMiner and BeatDB v1. Given the major differences between the design and frameworks used for computation, the BeatDB v1 output data looks different than the PhysioMiner output data, but there are valid explanations for most of the differences.

Given the cost of repopulation, both monetarily and temporally, we are comparing two resultant datasets that already existed (in the case of the PhysioMiner implementation, the large scale *beats* and *windows* tables already existed, with the aggregation table being created to match the parameters of the BeatDB v1 result file). On the left is the result of learning on the data from the NFS server that Dernoncourt used to develop BeatDB initially. The right side contains the results of learning on the output of the original implementation of PhysioMiner. These tests are meant to compare the effectiveness of learning algorithms on resultant datasets from the different iterations of BeatDB.

Upon inspection, there are a few noticeable differences. The biggest difference is that

Version:	BeatDB v1	PhysioMiner
Source:	NFS://beatdb/ABP-AHE_Classification_datasets/map60_segments_lag10lead60_trends.csv	dynamodb://beatdb_abp_windows_lead_60_lag_10_subwindows_10
Segment (rows):	629595	81760
Number of invalid rows (segments with errors):	0	1187
Features:	73	73
Data (non-id) features:	70	65
Number of AHE positive rows:	3275	2337
Percent of AHE positive rows:	0.52%	2.86%
10 trial AUROC mean:	0.8894	0.7962
10 trial AUROC std:	0.0027	0.0043

Table 3.1: Comparison of resultant datasets from BeatDB v1 and PhysioMiner.

PhysioMiner achieves an AUROC score 9% lower than BeatDB v1, which indicates a problem somewhere in the workflow. BeatDB v1 appears to generate significantly more segments than PhysioMiner, which may indicate that it was designed less efficiently in some way. This difference in design makes sense, seeing as PhysioMiner runs on AWS, which requires money to use, while BeatDB v1 used OpenStack and NFS servers already available at CSAIL for its distributed tasks. The large number of segments generated by BeatDB v1 may indicate that Dernoncourt did not skip over the condition window for the next rolling window entirely if the condition was detected in the condition detection algorithm. This may have an effect on AUROC and the number of AHE positive rows. (In the initial programming for the implementation of valid beat groups in PhysioMiner, discussed in 3.3.2, a similar mistake resulted in about 10 times more rows in the final dataset, making the process very costly and inefficient but providing giving evidence that this may have resulted in the large output table from BeatDB v1.)

Additionally, five data features are missing in PhysioMiner. The missing features are the five aggregation values for duration, since duration is calculated internally by BeatDB v1 and the aggregation step in PhysioMiner requires that the features are user-defined, meaning that PhysioMiner would not aggregate duration. PhysioMiner stores 5 extra variables (*beat_start_id*, *beat_end_id*, *condition_window_length*, *subwindow_index*, and *window_length*), which results in both datasets having 73 features with only 68 of the features (the 65 data features and 3 ID values) in common between the datasets.

These results prompted a direct comparison of the code and processes used in BeatDB v1 and PhysioMiner. Waldin's thesis gave significant background to the window finding algorithm, and analysis of it showed that PhysioMiner was lacking gap beat detection, meaning that some windows contained invalid sections of the waveform and should not have been included in resultant datasets. This could be one reason for the large difference in AUROC scores and is further explored in the next section.

3.3.2 Valid Beat Groups (VBGs)

In the port from BeatDB v1 to PhysioMiner, the condition detection step was changed, modifying the way windows are found over a waveform. The change is subtle, but seemingly has a large effect on the AUROC scores of the resultant dataset.

In BeatDB v1, condition detection runs the sliding window algorithm over contiguous segments of the waveform only. Contiguous segments of the waveform were easily identifiable because the beats database had a different schema, which created the *valid* field as an integer instead of a boolean. If *valid* had a value of 2, it meant that the beat was a gap beat, indicating that it had a duration of 6 seconds or more. Gap beats are likely the result of an error, either in the device or with the patient (removal of the detectors, etc. See 2.2.1 for more information) and were discarded from consideration. This allowed for the condition detection step to be more efficient and made the data more meaningful as it was guaranteed to come from signal rather

than noise. Additionally, the sliding window in condition detection had a size of $lead + lag + condition\_window\_length$ seconds, essentially bundling the condition window with the predictor (lag) window in the database. (This is a group window based on the definition and diagram in 3.1.2) If this group window concept is carried forward to future iterations of BeatDB, it could be used to do condition detection and feature aggregation simultaneously, cutting down on overall file processing time.

In the Java implementation, the *valid* field in the *beats* tables is a boolean, losing the capability to distinguish between gap beats and normal beats. Instead, the condition detection step starts at or near the beginning of the file, depending on the value of the *gap* argument, which can be used to start from a constant offset from the start of the waveform. The algorithm does not manually check beat duration for gap beats, so the windows formed come from the entire waveform, not just the contiguous beat sections. This could be the reason for the difference in AUROC score and database sizes between the resultant datasets from the Python and Java implementations of BeatDB, as the Java implementation used for testing includes invalid windows.

To test this idea, an instance of PhysioMiner's implementation from August 2016 and an updated version with valid beat groups implemented were compared. The result of linear regression on the aggregated windows tables from each version is shown in table 3.2.

One result from a learning trial for each version of BeatDB is provided to show sample output of the learning process used, but keep in mind that the valid beat group version did not always outperform the original version and the difference was not always as great as 2% in AUROC. However, the average over ten trials shows a clear increase by 1% in the VBG version, which may indicate that VBGs are necessary for BeatDB's overall accuracy. The standard deviation for the VBG version is smaller by a factor of 20, meaning that it is more consistent with its AUROC score than the version that does not look for gap beats.

Version:	PhysioMiner (original)	PhysioMiner (VBGs)
Sample files processed:	3000000.edf, 3000002.edf, 3000105.edf, 3900017.edf	
Rows:	150	160
Number of AHE positive rows:	30	20
Percent of AHE positive rows:	20.00%	12.50%
10 trial AUROC mean:	0.9699	0.9797
10 trial AUROC std:	0.0197	0.0096

Table 3.2: Comparison of original PhysioMiner implementation and PhysioMiner with valid beat groups (VBGs). This data was generated using a lead of 60 s, a lag of 10 s, and 10 subwindows, meaning that the predictor (lag) window was split into 10 sections for feature aggregation.

Why were VBGs left out of PhysioMiner when it is based off from BeatDB v1? Perhaps the *gap* argument was confused with the gap beat concept during the implementation of the Java port and the detail of the *valid* field for each beat object having three possible values was lost, preventing PhysioMiner from iterating over valid sections of the waveform while skipping invalid ones. Because of this, lead and lag were not required as arguments for condition detection, which resulted in the sliding window containing only the condition window with a duration of $condition\_window\_length$ seconds.

Not requiring lead and lag as arguments for condition detection forced the software to aggregate features by looking backwards in the raw waveform to find prediction windows. Instead, the software should have given the indices of the prediction window by coupling it with the condition window in the windows database. This could be the cause of significant inefficiencies. In future iterations of BeatDB, this must be considered, as the new requirement of lead and lag in condition detection could allow it to be performed simultaneously with feature aggregation while only considering contiguous sections of the waveform.

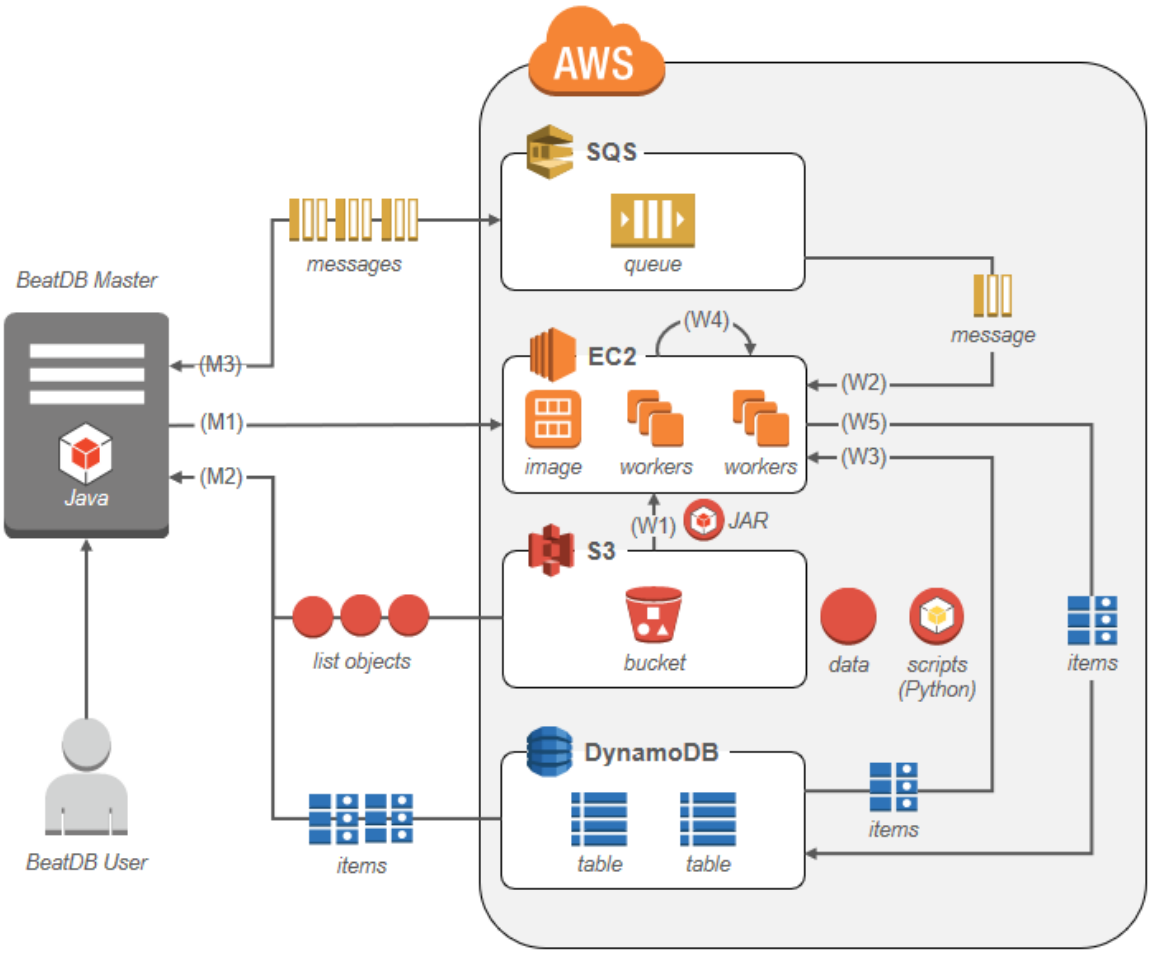


Figure 3-5: This figure, created by Alejandro Baldominos[4], shows the general PhysioMiner system architecture for both master and worker instances. Master instances start by creating worker EC2 instances and then gather items from DynamoDB in order to determine what needs to be processed. Once this has completed, the master instance uploads messages describing each task to the SQS queue. Worker instances spawn by receiving the PhysioMiner JAR file from S3 along with a message from the SQS queue, which prompts the workers to pull the specified items from DynamoDB and process them. Once this has completed, the workers write to the resultant DynamoDB table and continue to fetch and process messages until the queue is empty.

Chapter 4

BeatDB v3

BeatDB v3 is a new iteration of the BeatDB system, fully written in Python and designed with efficiency (correctness and performance), more complex feature extraction, and high code coverage as focuses of the project. It consists of a population/extraction stage and a condition detection/aggregation stage. Additionally, a standalone preprocessing stage, developed primary by Alejandro Baldominos, and a standalone file size reader were developed for use with BeatDB v3. This chapter will focus on motivations for creating a third iteration of BeatDB and then present a system overview along with a discussion regarding new features and optimizations made. Technical specifics for BeatDB v3, including how to run the software, implementation details, and other technical information, can be found in appendix A.

4.1 Motivations

Primary motivations for developing a new iteration of BeatDB v3 were initially related to extending the functionality of PhysioMiner. In order to add more flexibility to the system, developers wanted to add the capability to extract multi-channel and multi-sample/temporal features (where a chunk is a specific slice of the raw waveform (a beat would be considered a subset of a chunk)) and make it simpler for users to define their own signal types. However, given the large codebase and the amount of architectural complexity written within PhysioMiner, adding features to the system

would prove to be a lengthy and unclear process, made more difficult by the slow iteration time and the need for maintaining existing efficiency within the system.

On top of this, PhysioMiner does not result in datasets with AUROC scores similar to those generated from the original version of BeatDB. This issue was discovered through experimentation between the two versions (discussed in section 3.3.1) and was found to have a connection to the condition detection step (discussed in section 3.3.2). As a result of these issues, along with AWS issues that led to large efficiency issues and costs (see section 4.1.2), primary motivations for developing a new iteration of BeatDB became usability for both developers and non-technical users, efficiency, and correctness. These motivations will be discussed in terms of the shortcomings of the PhysioMiner system.

4.1.1 Usability

Although PhysioMiner was the first iteration of BeatDB intended for use by researchers that are not necessarily developers, some of the design decisions made in the final system are not particularly user-friendly. PhysioMiner did not use a configuration file and required that the user pass in all arguments via the command line. This gets particularly messy for complicated stages. For example, if a user wanted to fully process a set of raw data files (that is, detect beats from each raw data file and extract features from them, and then perform the sliding window algorithm to detect condition and aggregate features), the user would need to run the following commands with the required arguments listed. The commands tell PhysioMiner to run population/extraction, condition detection, and aggregation, respectively.

```
java -jar main.jar populate --bucket beatdb --table beatdb -n
    1 --features abp_features.txt --folder data --signal-
    type ABP --initialize-tables --key-name sarivera --iam-
    profile sarivera_beatdb
```

```
java -jar main.jar find --bucket beatdb --table beatdb -n 1
      --lead 60 --lag 30 --conditions abp_conditions.txt --
      initialize-tables --key-name sarivera --iam-profile
      sarivera_beatdb

java -jar main.jar aggregate --bucket beatdb --table
      beatdb_sar_local -n 1 --lead 60 --lag 30 --windows 10
      --aggregators aggregators.txt --features crest,diastole
      ,kurtosis,map,mean,pressure_area,pulse,rms,skewness,std
      ,systole,systole_duration,diastole_duration --
      initialize-tables --key-name sarivera --iam-profile
      sarivera_beatdb
```

By the end of the last command, the user would have generated a table in DynamoDB with subwindows, each with aggregated features, that can be used for machine learning. Given the large number of required inputs, using a configuration file for BeatDB v3 will significantly cut down on the complexity of these commands, especially since PhysioMiner is not resilient to unrecognized inputs being passed in and will quickly fail if that situation occurs. With BeatDB v3, the command to run the entire BeatDB system on a set of raw data files could be (and is) as simple as *python main.py create_dataset* with an optional $--config$ flag that can be supplied if the config file has a different name than $config.yaml$. Additionally, if unused arguments are listed in the configuration file, BeatDB v3 does not fail to execute, as PhysioMiner would when it is initialized with unfamiliar arguments.

Though PhysioMiner is not prohibitively complicated to run, it still requires some working knowledge of AWS and coding fundamentals. In order to practically use the system, the user has to have some idea of how AWS works (particularly S3, SQS, DynamoDB, and EC2) since PhysioMiner is highly dependent on these services. Small issues are likely to occur with usage that require the user to troubleshoot with knowledge of the AWS framework. Examples of issues that may arise include the SQS

queue having leftover messages from previous, incomplete runs and propagating them forward, EC2 instances failing during computations from silent memory issues, and the need to launch new workers with the correct boot script during computations.

The user is not able to get past this issue by running PhysioMiner locally, as its AWS functionality is interweaved throughout the design of the system. In order to run PhysioMiner locally, the user needs to spoof the existence of AWS services locally using the fake-s3[24], elasticmq[33], and DynamoDB Local[2] packages. Once these services are running and the BeatDB 'bucket' folder has been synced with fake-s3, the user can run PhysioMiner locally, limiting the amount of possible issues that can arise but also severely limiting the throughput of the system.

From the lens of a developer, PhysioMiner did not have high usability (code clarity) for researchers who wish to modify the Java source code. Significant chunks of the code existed as copy-pasted sections of other code, making it hard to modify the system and unclear what would happen when the system was modified. The design of PhysioMiner was particularly hard to understand, with each stage utilizing four classes for its internal structure. The master class would call the runner class to prepare each service, send messages, and initialize workers, with each process having its own set of logic and requirements. Given the lack of proper abstraction within the system, it became monolithic and was not easy to modify or adapt in a clean way. This usability issue was not necessarily true of the features and aggregation scripts, which were designed for simplicity and modularity in Python, but this design decision led to other issues (see the next subsection for details).

4.1.2 Efficiency

One of the major motivations for reimplementing BeatDB is efficiency. Though efficiency was a design goal for PhysioMiner, it was particularly inefficient in the way that it handled certain core algorithms, especially at scale. For example, in the population module, PhysioMiner takes in raw data, performs onset detection on the data

to parse the waveform into beats, and then validates and extracts features from these beats. This process could be completed in one loop over the beats array, but the implementation of population within PhysioMiner performs many more loops than necessary over the sample values of the beats and the resultant beats array. This problem of looping over large lists of values too frequently can be found in various stages of the PhysioMiner system and contributes to its overall inefficiency.

Additionally, the process of extracting features from beats is very inefficient due to its internal processes. As mentioned in the previous section, feature and aggregator functions were written in Python to make it easier for the user to define their own beat and aggregation features. This, combined with the way PhysioMiner gathers these functions, leads to a significant slowdown in processing. PhysioMiner would download all feature extraction Python scripts, perform them one by one on a beat by using subprocessing functionality in Java to call the Python scripts, and would read results by reading a set of temporary files in order to gather the values of the feature extraction functions. Subprocessing and file I/O always require a certain amount of overhead when utilized in a system, since the process running the code needs to suspend and hand control over to the subprocess, which then needs to write to file once it completes and hand control back to the original process. Because PhysioMiner needed to call Python scripts for its features and had to read each individual feature result from file, a lot of overhead was added to the population stage for each beat that was parsed. On top of this, each worker would delete the feature extraction scripts after processing a beat, forcing them to redownload the scripts before processing each new beat.

These issues, when taken together, caused significant inefficiencies throughout the system, as similar problems to these can be found in each step. In fact, the issue of downloading scripts and deleting them once a single beat has been processed is also found in the aggregation stage with the aggregation scripts. In order to fix these issues, future iterations of BeatDB need to pay close attention to the algorithms used and verify that they loop over large datasets as few times as necessary. Additionally,

future iterations of BeatDB should consider the efficiency issues that come with the addition or design of software features, as seen with the subprocessing slowdowns that the addition of Python scripts brought to PhysioMiner. To most easily mitigate this, BeatDB v3 will be written entirely in Python, so that no subprocessing or intermediary file I/O calls are necessary besides calls to the WFDB software package required for beat onset detection of specific signal types.

Algorithmic inefficiency is not the only efficiency issue that plagued PhysioMiner. Given the complexity of developing a distributed cloud system, PhysioMiner was bound to experience various framework-based inefficiencies when performing at scale. Issues with the AWS framework that led to overall inefficiency of the system will be discussed below.

AWS Issues

The algorithmic inefficiencies of PhysioMiner directly affected its performance when running the software in a distributed fashion over the cloud. Consider how the runtime and memory usage of PhysioMiner grow as the raw data file size grows, especially considering the algorithmic inefficiencies described above. Worker instances, which are inherently limited in size in order to keep resource costs low, are susceptible to runtime errors and memory issues as the program runs for longer amounts of time. When a worker instance runs into a memory error and halts its computation, there is no simple way for the master instance to receive knowledge of this and act accordingly. The user usually needs to manually inspect worker instances for logs to indicate their failure since these dead workers appear to be processing messages and are not easily discernible from functional worker instances. This inevitably leads to slowdowns within the framework, as less workers are available to work on messages as time goes on and lost workers may not be quickly noticed or replaced.

On top of this, some raw data files are too large for workers to process, leading to worker failure when these files are processed. In most cases, when the worker instance

stops working, the message the worker was processing will be sent back to the queue for reevaluation some time after the worker dies. This is intended to prevent files from being lost from consideration when workers fail while processing them, but leads to a big problem. If a worker dies because it received a message that points to a data file that is too large to process, the message will be readded to the queue, leaving another worker to grab it. This leads to a chain of broken workers, which amplifies slowdowns within the framework and wastes resources. This issue is further compounded when considering that worker instances must have at least as much memory as the maximum file size described in the message queue.

These issues, combined with the algorithmic and design-based inefficiencies of PhysioMiner, greatly affect the efficiency of PhysioMiner, which causes costs to skyrocket on large experiments. Future iterations of BeatDB need to focus on integrating within AWS in an efficient way and need to prevent the issues that arose with PhysioMiner related to silently dead workers and large files that were impossible to process on a single worker instance.

4.1.3 Correctness

As previously mentioned, a bug was found in the condition detection stage of PhysioMiner related to the detection of gaps and valid beat groups. Fixing this bug by considering gap sections and not considering windows containing them led to an increase in the 10-trial AUROC score for PhysioMiner (see section 3.3.2). Near the end of the development of BeatDB v3, a bug was found in the aggregation stage related to the generation of predictor (lag) window subwindows. This bug directly affected the calculation of aggregate features and led to an incorrect overlap of data.

When PhysioMiner is running feature aggregation, it reads the group windows from a *windows* table in DynamoDB and performs aggregation on the predictor (lag) windows found within the group windows. Before performing aggregation, the stage may split the predictor window into a number of subwindows, based on user input. This

concept is relatively straightforward, though the bounds of this design were never considered. For example, some aggregation features, such as those that perform linear regression on the value of beat features in a subwindow, such as the *trend* aggregate feature, may depend on each subwindow containing at least two beats. If a subwindow only contains one beat, feature functions like *trend* would fail to generate a meaningful result, as linear regression cannot give a result for a list of one value.

Given this, it makes sense that there should be a requirement that lag is some factor greater than the number of subwindows generated from the predictor window. However, PhysioMiner does not enforce this, and does not fail as expected when the number of subwindows is 10 and the lag is 10 s in duration (allowing for between 11 and 15 beats, on average), meaning that a *trend* value is still generated for each subwindow. Further investigation into this phenomenon revealed the existence of a bug in the process that generated the subwindow bounds. Because of an off by one error, the subwindows always contained an overlapping beat, meaning that, when the lag window is not sufficiently large enough to split into the requested number of subwindows, all subwindows generated will contain more beats than they should. In our example above, each subwindow contained at least two beats when they should have contained at least one beat. This is why the *trend* aggregate feature did not fail to generate a result in all cases when running PhysioMiner experiments where lag and subwindows were not compatible values.

This bug means that PhysioMiner tends to double count beats in subwindows, which would likely have a large effect on the aggregate features generated in subwindows that are supposed to have a small number of beats. This likely weakened the possible AUROC scores that PhysioMiner could achieve, and as such, should be avoided when developing the next iteration of BeatDB.

4.2 New Architecture

BeatDB v3 has a different architecture than previous iterations of BeatDB, but the base algorithms behind the stages of BeatDB remain the same. BeatDB v3's design is most closely related to the conceptual BeatDB design discussed in section 3.1, containing two stages that perform similarly to the Beat Object Generation and Dataset Preparation modules.

4.2.1 System Overview

As mentioned, BeatDB is split into two stages: population/extraction and condition detection/aggregation. These stages can be run independently or sequentially via the joint stage titled *create_dataset*. When running the modules independently, the population/extraction stage creates an intermediate .npy file containing the array of chunks generated from the stage and stores it in a path specified in the configuration file. This allows the condition detection/aggregation stage to be run directly after by reading the .npy files placed in this directory from the population/extraction stage, generating a windows and subwindows .csv file ready for machine learning analysis. These stages can be run immediately sequentially from within the code by running BeatDB v3 with the create dataset stage, which will only produce the windows and subwindows .csv files and is equivalent to running population/extraction followed by condition detection/aggregation. The independent method of running the stages may prove to have useful applications in cloud applications regarding BeatDB v3, but for all intents and purposes, it is faster and simpler to run the joint create dataset stage on raw data.

Assuming the user has set the proper arguments in the configuration file, the stages are performed as follows. For more detailed specifics about how these stages are run, see appendix A. For a simplified diagram of the BeatDB v3 system overview, see figure 4-1.

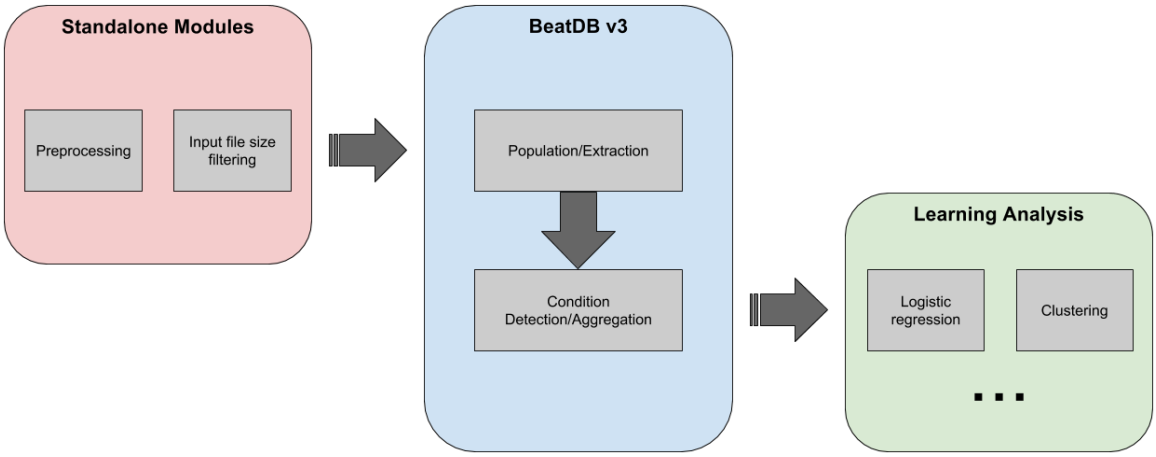


Figure 4-1: Simplified version of the data pipeline in BeatDB v3. Input data can be fed either directly into BeatDB v3 or into the standalone modules, with preprocessed output going to BeatDB v3. Population/extraction is always run first within BeatDB v3 and is followed by condition detection/aggregation. These stages may be run immediately sequentially or in parts, with the population/extraction module generating an intermediate file for later analysis with condition detection/aggregation. Lastly, the user may use learning algorithms on the resultant dataset from BeatDB v3's condition detection/aggregation stage to generate metrics about the data.

Population/Extraction

The first stage of BeatDB v3 is almost exactly the same as the population and feature extraction stages from PhysioMiner that were capable of being run sequentially and separately. BeatDB v3 processes the raw .edf data files found in the input directory one by one, referring to the *edf_to_chunks* method of the signal type class representing the signal type of the raw data to determine how to parse the raw data into chunks. For ABP, this consists of parsing the signals and pulses from the WFDB software for the raw data file. Once these have been found, the ABP signal type class will use these arrays to extract beats from the raw waveform. As it extracts a beat, it immediately calculates the features for that beat using the user-defined feature functions, allowing BeatDB v3 to only ever need to loop through the beats array when it is created. Users can compute rolling window features when performing this step over the waveform, and if the signal type contains multiple channels of data, can also define multi-channel features to be calculated for each chunk. Once all chunks have

been extracted, the chunks array is either exported to an .npy file in the intermediate file path if the population/extraction stage is being run independently, or the chunks array is passed to the next module directly if the population/extraction stage was run via the joint create dataset stage.

Condition Detection/Aggregation

The second stage of BeatDB v3 combines the condition detection and feature aggregation stages of PhysioMiner into one stage. The reason these stages were combined relates to the modifications made to PhysioMiner, documented in section 3.3.2. Since the modifications required that the lead and lag arguments be given as input to the condition detection stage along with the aggregation stage, and since the stages could be flattened into one loop as feature extraction was in the population/extraction stage discussed above, it made sense for efficiency and design purposes to combine these stages.

As with the previous stage, this stage processes chunks representing raw data files one at a time. Depending on how this stage is run, it either reads from the intermediate data path where the .npy files from population/extraction are stored in order to generate the chunks when run independently or starts with the chunks array already defined in the joint create dataset stage. Once the chunks array for a raw file has been created or passed in, the stage passes it to the *detect_conditions* function which performs a rolling window algorithm most closely related to the one used in BeatDB v1 over the chunks array. To recap, this rolling window algorithm works by starting at the beginning of the raw waveform and attempting to define bounds for the group window starting at this point, defining the *lead_start_index*, *condition_window_start_index*, and *group_window_end_index* from indices of chunks in the chunks array. If there are gap beats in the group window, the start index of the next group window is the first valid beat after the gap beat. If there are no gap beats in the group window, then the algorithm tries to classify the condition window (from *condition_window_start_index* to *group_window_end_index*) us-

ing the condition function.

Immediately after, the algorithm splits the predictor (lag) window into a number of even subwindows specified by the user (the subwindows are not always able to contain the same number of chunks, but are made as balanced as possible). Once the predictor window has been split into even subwindows, the algorithm aggregates features over each subwindow. After this, the algorithm creates an object to represent the group window, including the condition classification and the subwindows, which each contain their aggregate features in their own object abstraction, and moves on to the next group window position, which depends on the classification value.

If the classification is True, meaning that the condition window for the group window was determined to have the condition, then the rolling window algorithm places the start index of the next group window immediately after the end index of the positive condition window and continues. If the classification is False, the algorithm moves the rolling window ahead by 10%. This style of jumping (the amount of the waveform that the rolling window slides ahead by) is called 'v2 style' because this is the type of movement that was used in PhysioMiner. For completeness, 'v1 style' jumping, where the group window start index moves ahead by the length of the lag window, was also tested. A discussion regarding the differences between these styles of jumping can be found in section 5.1.1.

This rolling window algorithm is run until the end of the chunks array is reached, usually because the next group window would pass the end of the chunks array. Once all group windows have been analyzed, they are written to file. For simplicity of testing, all subwindows are also written to a separate file, containing all aggregated features and classification values as well (included for testing purposes). This process is done for each chunks array .npy file in the intermediate file path or for every raw data file passed in, depending on if this stage is run independently or in the joint create dataset stage, respectively. Pseudocode for the rolling window algorithm (called *detect_conditions* in the software) can be found in algorithm 1 in appendix

B. Examples of the different jump styles can be seen in the same pseudocode in lines 14 and 15.

4.2.2 Optimizations from Previous BeatDB Iterations

As mentioned in the motivations section, one of BeatDB v3's major focuses was on efficiency. As such, developers sought to make the algorithms behind BeatDB v3 run as quickly as possible. One way to do this involved fully designing the algorithms before implementing them. By trying to limit the number of loops over data structures, get as much data as possible in one iteration, and keep the codebase small and necessary, we were able to make the algorithms run in as minimal time as possible without applying more internal and low-level optimizations. For example, in PhysioMiner, a frequently used utility function to determine the first beat that occurs after a specified amount of seconds was incredibly inefficient because it always started from the start of the chunks array, meaning that the runtime of the utility function got worse and worse as the number of chunks in the waveform got larger. By recognizing these issues and the unclear way in which they can propagate, we can design around them and truly limit the number of loops required to compute specific stages. Now, both stages only need to perform one loop over the input data, with condition detection/aggregation containing some smaller loops internally in every iteration to deduce beat indices based on time durations.

We also made sure to not make redundant steps, such as downloading scripts and deleting them after applying them on a single set of samples or rederiving constant values like feature name to function dictionaries within loops. For example, the population/extraction algorithm only needs to do one loop over the beats array to perform feature extraction once the beats have been parsed from the raw waveform and does so with feature functions that are included within modules to the system. Compare this with PhysioMiner, which, for each chunk, has to download the feature scripts individually, run them via a subprocess, parse the results using file I/O libraries, and then delete the feature scripts. By writing the procedures so that they try to do

all required data generation at once and create all frequently used constants before iteration, the algorithms were able to minimize the number of calculations necessary for processed files.

In the same vein of implementation based optimizations, BeatDB v3 utilizes a higher-order function paradigm for its feature extraction and feature aggregation components. This is done by creating a *Features* class in the code that contains static methods representing the feature and aggregation functions used by BeatDB. When the users arguments from the configuration file are parsed, the system will generate a dictionary mapping feature or aggregator names to their respective functions in the Features instance the raw data and signal type indicated in the configuration file relate to. These functions are then used by the stages for efficient extraction and aggregation, as the functions are already in memory and all consist of constant time calculations. The higher-order function paradigm used by BeatDB v3 allows for efficiency in the extraction and aggregation procedures and keeps the design clean and easy-to-understand for users hoping to modify the source code.

BeatDB also sought to achieve optimizations directly related to its design and the structure of the code base. The process of running each of the stages of PhysioMiner contained abstract classes that dealt with the master/worker split, which is likely a result of its implementation being very intertwined with the AWS architecture. Since BeatDB v3 was initially developed as a local application, its structure is more direct and less abstract but is still flexible enough to allow for multiple cloud frameworks to be supported without needing to write more code than necessary to support the framework's API. Another benefit of this development approach relates to the efficiency of the local version of BeatDB v3. The local implementation of PhysioMiner could have been more efficient if it did not require the spoofing of AWS services, since the code could utilize the power of the computer running the process without needing to worry about the overhead of communication between services. Developing BeatDB for local use first gives us the guarantee that, even if a cloud based framework were ported in and failed for whatever reason, the local version of BeatDB will still perform

well, allowing users to completely and quickly ditch the cloud framework instead of relying on fake services for the code to work locally.

By reducing the number of stages of BeatDB from four (in PhysioMiner) to two, we were able to cut down on some of the overhead that was created by needing to run the stages manually. Additionally, the development of the joint create dataset stage allows for users to parse raw data directly into the final output of BeatDB v3, which is the most efficient way to use BeatDB v3 since the joint stage does not need to write to file between the population/extraction and condition detection/aggregation computations. The smaller number of stages could potentially cause issues when processing larger data files, since workers would need to perform what was two stages worth of computations in PhysioMiner in one stage. This may limit the size of files that are able to be computed by BeatDB v3, but it is likely that the computations between stages of PhysioMiner and BeatDB v3 are similar in resource usage, since the input data and algorithmically generated data is the same between systems.

4.2.3 Additional Functionality

BeatDB v3 was not designed as a subset, but a superset, of PhysioMiner, offering new ways of calculating features from multi-channel data and standalone utility modules for use before execution of BeatDB v3 on raw data. This section will go into more detail about each of these new features.

Multi-Sample and Multi-Channel Feature Extraction

PhysioMiner requires the usage of ABP or ECG waveform data (written in .edf format). In order to allow for the most flexibility related to signal types, BeatDB v3 has a clear and concise schema for signal type classes that allows for many kinds of signal types to be represented, whether they contain beats or chunks or contain multiple or single channel data. It also allows for researchers compute multi-channel and multi-sample features from specific types of physiological data.

Multi-channel features are features that are computed using multiple channels of signal data. This is dependent on the signal type that is being analyzed. Some signals, such as ABP, are single-channel, and as a result, cannot generate multi-channel features unless the signal is included with others and the signals are synced. Multi-sample (temporal) features are features that are generated by a rolling window algorithm over a number of consecutive beats. multi-sample features are not generated until at least some number of chunks, set by the *rolling_window_length* argument in the population/extraction stage section of the configuration file, have been seen previously in the population/extraction stage while processing of a specific raw data file. Because of this, multi-sample features can technically be generated from any signal type as long as the signal type generates enough chunks from the raw data file. However, multi-sample features will only have a value for the chunks that occur after the $rolling\_window\_length^{\text{th}}$ chunk in the chunks detected in population/extraction, since the rolling window algorithm requires that at least this many chunks have been seen previously to generate a multi-sample feature.

Allowing for this type of feature generation has potential to strengthen the accuracy of the datasets generated by BeatDB and allows researchers to test the correlation of previously untestable features with classification of specific conditions. Specific information related to the implementation of multi-channel and multi-sample beats in BeatDB v3 can be found in section A.3.

Standalone Utility Modules

To prevent the issues encountered with PhysioMiner and AWS, a filter for BeatDB was implemented that allows users to exclude raw data files that are outside of a given file size range from consideration when running the system. This should help prevent issues with 'killer' messages discussed in the previous section. These are shown in the system diagram for BeatDB v3 in figure 4-1. Though this feature was not integrated within BeatDB, users can still use it to analyze their data before processing it with BeatDB v3, potentially saving them from issues caused by tiny and massive file sizes

and limited worker instance memory.

Additionally, a standalone preprocessing module for BeatDB was developed, allowing for users to quickly preprocess their raw data with user-defined functions before running it through BeatDB. This module, developed by Alejandro Baldominos, is usually used to remove noise from the signal before processing the data, but could be used for other kinds of user-defined preprocessing as well. This preprocessing module was the first module for BeatDB v3 developed, with later modules making use of the AWS software patterns Baldominos introduced in preprocessing.

Chapter 5

Demonstration of v3 with Original ABP Data

5.1 Data-Based Comparison Between PhysioMiner and BeatDB v3 Versions

In order to demonstrate the effectiveness of the BeatDB v3 system, average run-times were recorded and 100-trial AUROC scores were computed from the final output of two versions of PhysioMiner and two versions of BeatDB v3. The two versions of PhysioMiner are the original implementation of PhysioMiner and the modified version of PhysioMiner that includes VBGs. For more discussion about these versions of PhysioMiner, see section 3.3.2. This chapter will discuss the difference between these versions of BeatDB v3 and the experiment, attempting to draw conclusions from the results.

5.1.1 v3 Jumping Styles

In all iterations of BeatDB, the rolling window algorithm in the the condition detection stage places the start of the next rolling window right after the end of the current one if the current condition window is classified as having the condition. However, when the condition window is classified as not having the condition, the amount of

distance the rolling window moves by is dependent on the iteration of BeatDB that is being run. (For a programmatic example of these styles of jumping, see algorithm 1 in appendix B on lines 14 and 15.)

In BeatDB v1, the condition detection stage moves the rolling window forward by the duration of the lag window, meaning that the start index of the next rolling window is the start index of the lead window in the current rolling window. Since lag values are usually small, this process results in a significant amount of windows being produced. This was not an issue for BeatDB v1, which was run on free OpenStack servers in CSAIL.

For PhysioMiner, the condition detection stage moves the rolling window forward by 10% of the entire *chunks* array for the file being processed. This style of jumping is more efficient, but has the possibility of ignoring large sections of chunks if the file or individual chunk size is very large. Because this design decision was made for efficiency, its effect on BeatDB's resultant AUROC scores was unknown. For efficiency, BeatDB v3 will use v2 style jumping, but the source code can easily be modified to use v1 style jumping (see algorithm 1 in appendix B) which makes it easy to test the effect of jump style on resultant AUROC scores. To understand how the chosen style of jumping truly affects BeatDB and how PhysioMiner and BeatDB v3 compare, we will analyze AUROC scores and run-time of the original implementation of PhysioMiner, the modified VBG implementation of PhysioMiner, BeatDB v3 with v2 style jumps, and BeatDB v3 with v1 style jumps.

5.1.2 Experiment and Results

In order to properly test these iterations, a common dataset with common arguments needed to be determined. In the process of picking a sample set of arguments, the bug discussed in section 4.1.3 was discovered. In order to get around this issue, a lead of 60 s, lag of 30 s (instead of 10 s as used in section 3.3 when comparing PhysioMiner iterations, see 4.1.3 for details regarding why picking a lag duration similar in num-

ber to the number of subwindows is not a good way to represent the data), and 10 subwindows was picked, since each subwindow would be expected to have at least 3 beats most of the time. This allows the aggregated features to be more representative of the condition and prevents aggregate features such as *trend* from having invalid values, as would occur if some subwindows contained only one beat. In layman's terms, these parameters mean that we are attempting to detect a condition found in a 30 minute window 1 minute after a point t using 30 seconds of data before point t. For a table of inputs and parameters, see table 5.1.

Sample files processed:	{3000000, 3000002, 3000105, 3900017}.edf
Lead:	60 s
Lag:	30 s
Number of subwindows:	10
Condition window length:	1800 s
Condition:	AHE (acute hypotensive episode)
Condition definition:	a window has AHE if 90% of beats in the window have a mean arterial pressure (MAP) feature value less than 60 (our threshold)
Chunk-level features:	Crest, diastole, diastole duration, kurtosis, mean arterial pressure (MAP), mean, pressure area, pulse, root mean square, skew, standard deviation, systole, and systole duration
Aggregate features:	Kurtosis, mean, skew, standard deviation, and trend

Table 5.1: This table shows common parameters used for experimentation in this chapter, which compares versions of PhysioMiner and BeatDB v3. Chunk-level and aggregate features indicate which functions should be performed on either the samples that comprise a chunk or specific feature values for chunks in the same group window, respectively.

The same sample files used to compare PhysioMiner iterations in section 3.3.2 were used for this experiment. As was done with the results from section 3.3, metrics such as AUROC scores in the table are generated by taking the average metric score over 100 trials using logistic regression with a test/train split of 0.6. All average run-times are recorded over five trials of the described process being run. Additionally, because it was not possible to run all stages immediately sequentially in PhysioMiner, the

'All Stages Avg. Run-time' for PhysioMiner versions is a sum of the averages of the run-times above it and does not have a standard deviation value. Other special cells include run-times for population and extraction, which are the same between versions of the same iteration since version differences are only in the condition detection procedures, and run-times for condition detection and aggregation for BeatDB v3, since these stages are not separable in BeatDB v3. The results can be seen in table 5.2.

Version:	PhysioMiner original	PhysioMiner VBGs	BeatDB v3 v2 style jumps	BeatDB v3 v1 style jumps
Number of subwindows (rows):	150	160	250	7670
Number of AHE positive rows:	30	20	20	20
Percent of AHE positive rows:	20.00%	12.50%	8%	0.261%
Population & Extraction Avg. Run-time	6841 ± 460 s (114 min)		94.24 ± 0.66 s	
Condition Detection Avg. Run-time	70.70 ± 9.66 s	242.9 ± 12.7 s	9.041 ± 0.12 s (stages joint in v3)	133.1 ± 2 s (stages joint in v3)
Aggregation Avg. Run-time	2854 ± 103 s (47.57 min)	2741 ± 151 s (45.68 min)		
All Stages Avg. Run-time	9765 s (162.75 min)	9825 s (163.75 min)	99.92 ± 0.9 s	225.4 ± 4.5 s
All Stages Avg. Run-time Scale	1.0x (baseline)	0.9939x	97.73x	43.32x
Accuracy mean:	91.2 ± 3.38%	94.4 ± 2.78%	97.5 ± 1.54%	99.6 ± 0.08%
F1 mean:	94.8 ± 2.06%	96.7 ± 1.63%	98.7 ± 0.85%	99.8 ± 0.04%
AUROC mean:	94.8 ± 2.55%	97.3 ± 2.11%	98.5 ± 1.33%	99.1 ± 0.51%

Table 5.2: Comparison of the original PhysioMiner implementation, the modified VBG version of PhysioMiner, BeatDB v3 with v2 style jumps in condition detection, and BeatDB v3 with v1 style jumps in condition detection, all run locally on an xl.12core OpenStack instance with a lead of 60 s, a lag of 30 s, and 10 subwindows. Average run-times were recorded over five trials. Accuracy, F1, and AUROC metrics are taken over 100 trials of logistic regression with a test-train split of 0.6.

The results from the experiment show promise for BeatDB v3. Both versions of BeatDB v3 attain higher AUROC scores over both versions of PhysioMiner. Additionally, BeatDB v3 with v2 style jumping achieves approximately 100 times speedup over PhysioMiner when running the entire BeatDB system on the sample raw data files, which total about 157 MB. Processes that took PhysioMiner almost three hours to complete took BeatDB v3 only two minutes to complete.

How did the jumping styles compare? As expected, BeatDB v3 with v1 style jumps generated a significant amount of windows (7420 more windows than BeatDB v3 with v2 style jumping). While this allows BeatDB v3 with v1 style jumping to achieve an AUROC average that is .5% higher, the system took more than twice as long to run than BeatDB v3 with v2 style jumping. Perhaps there is a middle ground between window generation and time to process that increases AUROC score while keeping run-time and window generation relatively low. Additionally, accuracy and F1 scores consistently increase along all iterations of BeatDB. The accuracy score represents the accuracy with which each system classifies the condition windows, while the F1 score represents the accuracy of the system in classifying condition-positive samples. The fastest iteration, BeatDB v3 with v2 style jumps, achieves an accuracy score of 97.5% and an F1 score of 98.6%. The most accurate iteration, BeatDB v3 with v1 style jumps, achieves optimal accuracy and F1 scores with 99.6% and 99.8%, respectively. One would assume BeatDB v3 with v1 style jumps might have a high accuracy because of the large number of condition-negative subwindows that are generated with that method of jumping, but because the F1 score for this iteration is even higher, this is verification that the iteration not only correctly classifies negative windows but the small number of positive windows as well. Both iterations of BeatDB v3 are more accurate than the iterations of PhysioMiner, but the iteration with v1 style jumps performs amazingly well with condition detection, correctly identifying nearly all windows in the dataset.

Since this experiment was somewhat small in file size, it is not fully known when BeatDB v3 with v1 style jumping would start to see memory issues related to the

number of windows it generates in its rolling window algorithm. Future researchers should be cautious of the file sizes of the input data if they choose to use this style of jumping. Perhaps there is value in exploring the usage of different jump values in the condition detection algorithm, as different jump values may directly impact the computability of a raw data file on a worker instance.

Overall, BeatDB v3 performs between 1 and 4 percentage points better when comparing average AUROC score with PhysioMiner and manages to do so while significantly reducing the amount of processing time. More experimentation needs to be done in order to correctly classify BeatDB v3's performance on cloud based systems and other signal types. Despite this, BeatDB v3 is quite promising in its ability to process raw data quickly and accurately and has done so while maintaining a small and easy-to-understand code base.

5.2 Comparison of BeatDB Iterations

Given the results in the previous section and section 3.3, it can be shown that successive iterations of BeatDB have consistently managed to increase the predictive accuracy of the generated dataset, with BeatDB v3 achieving the greatest accuracy and AUROC scores in comparison to all previous iterations. To better understand how the BeatDB system has grown over the development of its iterations, we will directly compare each consecutive iteration of BeatDB. This will give a better understanding of the improvements made at each step and how BeatDB's usability (in terms of ease of use and code clarity), extensibility, performance (in terms of run speed and accuracy), and service (framework) cost were affected by each implementation.

5.2.1 BeatDB v0 and BeatDB v1

The differences between BeatDB v0 and BeatDB v1 are quite obvious. BeatDB v0 only contained a population/extraction module, whereas BeatDB v1 also contained condition detection and aggregation modules. Despite this, BeatDB v0 and v1 were

very similar in design, with BeatDB v1 possibly reusing the same code for the population/extraction module. Given this design influence, BeatDB v0 and BeatDB v1 achieved comparable performance for the population/extraction modules and were developed with similar use-cases in mind, with each iteration hardcoded to run on the specific environments used for development. Both of the theses that introduced these iterations of BeatDB focused more on the analysis of the output datasets than the implementation of the systems. This focus inherently limits the extensibility of these systems as they were primarily designed to generate results for experimentation, not for system maintainability, ease of adding new features to the software, or usability for a general audience. Though their use-cases were limited and each version was not easily customizable, the iterations were free to run since they utilized free OpenStack servers located at CSAIL for the bulk of their computations.

5.2.2 BeatDB v1 and PhysioMiner

PhysioMiner attempted to make BeatDB v1 more user-friendly and cutting-edge by implementing the system within the AWS framework. The AWS framework was chosen for its potential to increase performance and the experience that would be gained by implementing the software in a then-new cloud framework. PhysioMiner allowed for deeper user customization, allowing the user to directly specify arguments to each stage and control how their data was processed. The user also had a much simpler way to implement their own feature and condition functions, writing them in individual Python files for the system to read. Though users were encouraged to define their own signal types, the task of writing a signal type class containing all required logic was messy and hard to understand. Given that the stages were all designed to be run on the AWS framework in sequential order while processing files in a parallel fashion, PhysioMiner provided further scalability potential than BeatDB v1. This was because of the ease of scaling when using AWS (provided the user is willing to spend money on resources) and because BeatDB v1 did not allow for the processing of the more granular tasks that PhysioMiner could parallelize by nature of its design, since it split its main pipeline into four separately runnable stages instead

of two.

Unfortunately, this potential was not fully realized, as PhysioMiner was plagued with bugs, especially when processing large data files at scale. As a result, these bugs caused large amounts of resources to be used for faulty computations that could not be detected as faulty until after computation was completed. These issues were largely related to queue timeout issues, which were exacerbated by worker instances silently failing when processing specific 'killer' messages. BeatDB v1 did not possess the kind of distributed power that PhysioMiner did, but it was still free to run and could be run on files in parallel, making BeatDB v1 ultimately better for large-scale experiment runs. Still, PhysioMiner achieved higher AUROC scores and significantly higher accuracy scores over BeatDB v1, though resource concerns are typically more important to researchers than relatively small increases in resultant AUROC scores.

5.2.3 PhysioMiner and BeatDB v3

The issues with resource usage and low performance within PhysioMiner made these concepts a focus in the development of BeatDB v3. Additionally, the feature creep and lack of proper abstraction in certain parts of the software made PhysioMiner monolithic and difficult to maintain, especially for researchers unfamiliar with its system design. This prohibited users unfamiliar with the source code from tinkering with it, limiting the extensibility of the system. Lessons learned from PhysioMiner's design influenced the development of BeatDB v3 in a positive way, aiming to make the code more understandable, documented, and concise in order to encourage other researchers to develop on top of the system. In BeatDB v3, code readability is achieved by using superclass patterns for cloud framework components, using the same design pattern for all stage implementations, restricting code duplication as much as possible, and documenting the code heavily, ensuring that all code segments are fully understandable. By making every step of the code clear to understand, the potential for extensibility of the software goes up, as a wider range of users will be able to understand and modify the source code to suit their needs.

In terms of more general usability, BeatDB v3 has been designed for ease of use, making it simple to run the pipeline on raw data even if the user is not very familiar with coding. BeatDB v3 has a config file that contains all run-time arguments, making it very easy to for a new user to generate output immediately with a simple one-line command. See section 4.1.1 for a more detailed explanation of how BeatDB v3 approached usability concerns.

BeatDB v3 sees a very large increase in performance, generating a machine learning dataset from raw data by a factor of 97.73 times faster than PhysioMiner. As an example, if PhysioMiner took an hour to process a file from the beginning of the pipeline to the end (population to feature aggregation), BeatDB v3 would take a little under 36 seconds to process the same file in its local mode. This whopping increase in performance will allow resource costs to stay minimized and, if the cloud framework components are carefully designed, can prevent the same worker and queue issues from PhysioMiner from having an effect on BeatDB v3.

Version:	BeatDB v0	BeatDB v1	PhysioMiner	BeatDB v3
Ease of use:				
Code clarity:				
Extensibility:				
Run speed:				
Accuracy:				
Service cost:				

Table 5.3: Simple comparison of BeatDB iterations. PhysioMiner has a lighter shade of red for performance because, while it achieved faster performance than previous iterations for smaller cases, it had serious issues with data in large scale applications, causing performance to slow significantly as workers silently failed while costing hundreds of dollars per day of computation (see section 4.1.2 for more details). PhysioMiner also has a lighter shade of green under accuracy since it managed to increase average AUROC scores on a small dataset but achieved lower AUROC scores on the overall dataset when compared to BeatDB v1 (see section 3.3.1 for more details).

Chapter 6

Conclusion

BeatDB v3 has been shown to achieve higher average AUROC scores than PhysioMiner while increasing software processing speed by a factor of almost 98. This is a great achievement for the system, but further improvements can still be made on BeatDB v3. This section will discuss how our research questions from the first chapter were met, along with future work related to the BeatDB system.

6.1 Research Findings

To discuss the findings of this research, the research questions presented in section 1.2 will be answered given the results of the research and work on the BeatDB v3 system.

Does the PhysioMiner system achieve comparable correctness to the BeatDB v1 system?

As shown in section 3.3, initial comparisons of BeatDB v1 and PhysioMiner (original) on the same input showed large increases in accuracy, F1 score, precision, and recall for PhysioMiner along with a 9% decrease in AUROC score. Attempting to rewrite condition detection in PhysioMiner with regards to valid beat groups, as discussed in section 3.3.2, led to slightly higher AUROC scores, but did not reach the AUROC

scores that BeatDB v1 did. This, along with compounding issues with the PhysioMiner source code, prompted the design and implementation of BeatDB v3, which will be the focus of responses to the remaining research questions in this section.

How can the BeatDB system be more usable?

Usability was one of the key motivations for the development of BeatDB v3, which aimed to create a very user-friendly iteration of BeatDB. This was achieved by focusing on code clarity and ease of use. Users that are not very familiar with computer science can run the BeatDB v3 code with relative ease, given the short commands necessary to initiate the pipeline and the grouping of all input arguments in the configuration file. Users that wish to modify the source code will find it easy to navigate through the flow of the software, since every function is well documented and the code heavily relies on object oriented principles to minimize code duplication. This makes it much more simple for users to implement their own signal types and support for different cloud frameworks while retaining the simplicity of defining the feature and aggregator functions found in PhysioMiner. For a more in depth treatment of this topic, refer to section 4.1.1.

How can the BeatDB system be more efficient, in terms of performance, software design, and implementation?

As discussed in section 4.1.2, BeatDB v3 was rigorously designed before implementation in order to achieve the minimum number of loops required over the data and within the software. Learning from the mistakes of PhysioMiner provided the BeatDB v3 development team with valuable insights related to performance and software design. This directed the team toward employing a higher-order functions paradigm for feature and aggregator functions, coupled with object oriented abstraction found throughout the software in order to reuse code without duplication. Beyond these large data abstractions, attention was given to each core and utility algorithm necessary for running stages of BeatDB in order to minimize the run-time of each algorithm

and cut down on the overall run-time of the system. Given the 97.73x increase in processing speed (see section 5.1.2), BeatDB v3 far exceeds expectations for performance.

Does the BeatDB v3 system achieve comparable correctness to the PhysioMiner and BeatDB v1 systems?

Section 5.1.2 discusses the outcomes of logistic regression trials on multiple versions of PhysioMiner and BeatDB v3 in more detail. BeatDB v3 manages to achieve an average AUROC score that is 1.2% higher than the greatest average AUROC score attained by all PhysioMiner versions. The accuracy and F1 scores are also much better for BeatDB v3 than PhysioMiner, with BeatDB v3 achieving scores that are 3.1% and 2% higher than PhysioMiner, respectively. When BeatDB v3 is allowed to consider more subwindows, usually because of a jump style that makes the rolling window algorithm move more slowly across the raw waveform (see section 5.1.1), it can achieve average AUROC scores that are higher than those achieved by PhysioMiner versions (see table 5.1). More research needs to be done to consider the optimal jump style that balances run-time and average AUROC score increase.

How can functionality be added to BeatDB v3 to allow for more customization, input filtering, and the ability to process multi-dimensional waveforms?

As mentioned in section 4.2.3, BeatDB v3 has added features that allow for easy implementation of multi-channel and multi-sample (temporal) features. Because of the higher-order function design, implementing multi-channel features is trivially different from single-channel features, while multi-sample features require slightly more expertise to create. For a more technical treatment of this feature, refer to section A.3. The development of BeatDB v3 also led to the development of the standalone modules for file size filtering and data preprocessing, which can be used with BeatDB v3 to further customize the output or prepare data for processing, respectively.

6.2 Future Work

Given the run-time of BeatDB v3 and its ability to implement multi-channel and multi-sample features, it is possible for researchers to analyze new and/or customized signal types in new ways with the system. Previously, researchers at ALFA and Philips had difficulty predicting hemodynamic instability (HDI), a condition associated with unstable blood pressure, because it is not a condition that is determined entirely by physiological readings. Perhaps the new multi-channel feature computing and chunking abilities of BeatDB will allow for new ways to analyze and understand aggregate HDI datasets.

Further development should be done on the various types of cloud architectures that BeatDB can run on. While BeatDB v3 is integrated within AWS, work should be done so that BeatDB v3 can also be run on OpenStack. Doing this would allow ALFA researchers to perform experiments on large amounts of data with small jump styles for free.

Speaking of jump styles from the condition detection procedure, more research should be given to the effect of different jump styles on AUROC average and run-time. It is possible that a jump style that maximizes AUROC score while minimizing run-time exists. If researchers can find this, they may be able to get the benefits of both v1 and v2 style jumping while avoiding the large run-time slowdown associated with v1 style jumps.

To make analysis of resultant datasets faster, developers could add a simple learning module to the end of BeatDB. Some complications of this include the way that final output data is stored by BeatDB v3. Since each subwindow table that is stored is placed in a folder named after each raw data file when BeatDB v3 is run locally, users need to combine subwindows tables in order to get a final, complete dataset of all subwindows for all raw data files. This can be done programatically but work has not been done on this task. Because the experiments with BeatDB v3 only used a small

number of sample files, file combining was done by hand for those trials.

Expanding upon the idea of incorporating learning into the system, BeatDB is one part of the large system that ALFA has conceptualized. BeatDB could be part of a software suite that allows simple machine learning on any raw data. Connecting various software systems related to machine learning into one massive machine learning framework is a long term goal for the ALFA group. Other projects developed within the group could be included in this suite, such as MLBlocks, which is a platform for quick and customizable learning of preprocessed data. Users could theoretically use BeatDB to process their data and MLBlocks to analyze their data in a pipeline similar to how population/extraction feeds its output to the condition detection/aggregation module. This goal requires that a significant amount of work is done on other projects related to this overall framework, since there may be interface clashes between the projects that would comprise this system. For example, the output of BeatDB would need to be formatted in such a way that MLBlocks can easily accept is as input. While grandiose, this large machine learning pipeline may prove to be a breakthrough in the way researchers perform machine learning experiments if it is able to combine the work of these projects in a cohesive way.

Appendix A

BeatDB v3 Interface

A.1 BeatDB v3 Code Structure

```
beatdb/
├── data/
├── representations/
│   ├── Beat.py
│   ├── GroupWindow.py
│   └── PredictionWindow.py
├── signals/
│   ├── abp/
│   │   └── abp.py
│   ├── eeg/
│   │   └── eeg.py
│   ├── abstract_signal.py
│   └── general_features.py
├── config.yaml
├── main.py
└── utils.py
```

Welcome to BeatDB v3's internals. Each code file should be well documented, so be sure to read docstrings and informative comments found in more confusing functions. BeatDB v3 is run by calling the *main.py* file, which contains functions necessary for parsing the configuration file and contains the main code for running each of the stages of BeatDB v3. Based on the user's command line argument for what stage to run, a specific function for the stage is called that parses its required variables and generates necessary constants to be used throughout the stage.

If the population/extraction module is being run, the function will parse necessary args from the configuration file and a $feature\_name\_to\_function\_dict$ will be generated and set as an attribute to the $signal\_type\_class$ file by using the utility function $set\_feature\_name\_to\_func\_dicts\_in\_signal\_type\_instance$. The dictionary maps feature names to functions, found in the $signal\_type\_class$ file for the $signal\_type$ specified by the user in the configuration file.

Signal type classes generally follow the structure presented in $abstract\_signal.py$. That is, each $signal\_type\_class$ should have an edf_to_chunks method that performs the main population/extraction algorithm (that is, parses the data into beats and extracts features from them). Additionally, each of the $signal_type_class$es should have a $determine\_validity$ static method that can be called on samples of that signal type. For ABP, this method implements the signal abnormality index mentioned in the thesis. Lastly, $signal\_type\_class$ should contain multiple static feature methods that represent possible feature functions to be used with that $signal\_type\_class$ in population/extraction. These feature functions can be defined directly or call functions from $general\_features.py$ to prevent code duplication. For ABP, edf_to_chunks calls $get\_samples, get\_pulses$, and $extract\_beats$, which yields the chunks array used in condition detection/aggregation. In ABP, Beat objects, defined in $representations/Beat.py$, are created and stored in the chunks array. If population/extraction was run independently, the chunks array will be stored as a .npy file in the $intermediate\_file\_path$ argument of the config file. If it was run in the joint $create\_dataset$ stage, the chunks array is immediately available for

condition detection/aggregation.

Condition detection/aggregation is also defined in *main.py* and, like population/extraction, consists mostly of configuration argument unpacking and generation of necessary constants, such as the $aggregator\_names\_to\_function\_dict$, which behaves similarly to the $feature\_name\_to\_function\_dict$ in population/extraction. After these variables are set, the stage loops over .npy files in the $intermediate\_file\_path$ in order to generate the chunks array and then calls the function $detect\_conditions$ to start the rolling window algorithm. This function is left in *main.py* because it is the main algorithm behind the stage. This algorithm, shown in a simplified version as algorithm 1 in appendix B, performs the rolling window algorithm described in chapter 3, checking for gap beats in the group window bounds. If no gap beats exist in the group window bounds, the algorithm calculates the classification and subwindows, along with their aggregated features (with each subwindow stored as a $PredictorWindow$ instance) in the same iteration and combines all this information in a $GroupWindow$ instance. These $GroupWindow$ objects are collected and written to file at the end of the stage.

Assuming the configuration file is the same as the one listed in A.5, the output folder would look as follows after running the joint create dataset stage. (The *abp_npy/* folder is an example of an intermediate data folder that would contain .npy files containing chunk arrays if the population/extraction stage was run independently.)

```
beatdb_output/
├─ <segment_id>/
│  ├─ <segment_id>_windows_lead_x_lag_y.csv
│  ├─ <segment_id>_windows_lead_x_lag_y_subwindows_z.csv
│  └─ <segment_id>_windows_lead_x_lag_y_subwindows_z_joint.csv
└─ abp_npy/
   └─ <segment_id>.npy
```

As shown in the directory tree above, three resultant files are generated from condition detection/aggregation. These files represent the $GroupWindow$ instances without subwindows listed, the $PredictionWindow$ instances with classifications from the parent $GroupWindow$ instances and pointers back to the parent $GroupWindow$ and the raw data file, and a joint file containing both of these instances in a staggered way. These outputs were included for variety, but the most useful of the output data files is $*segmentid*\_windows\_lead\_x\_lag\_y\_subwindows\_z.csv$. Groups of these files generated with the exact same arguments can be combined (without copying over the header) into a large file containing all subwindows generated with specific arguments. This large subwindows file can be used for learning directly, since all aggregate features are in the file and since classifications from each parent $GroupWindow$ are written in the file for each subwindow.

A.2 Running BeatDB v3

A.2.1 Local Mode

To run BeatDB v3 locally, place raw .edf files into the *data_path* specified in the configuration file. Once you have populated this path with raw data files, make sure that your configuration file has the proper arguments. If required ones are missing, BeatDB v3 will throw an error and point out the missing argument. Once the configuration file has been verified for correctness, open a terminal at the root of the *beatdb/* folder (shown in the first directory tree) and run the following command to run the entire BeatDB system on your raw data files, from population/extraction to condition detection/aggregation (if your configuration file is not named $config.yaml$, you will need to also supply a $--config$ argument with the name of your config file, including the file extension):

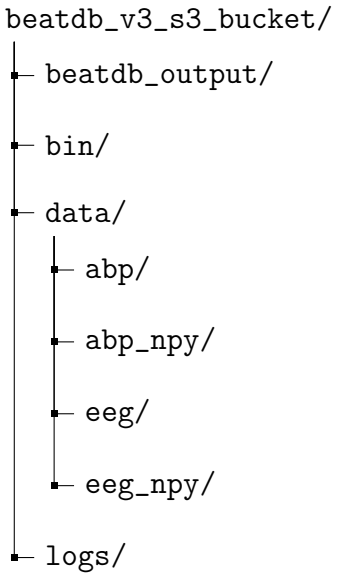
```
python main.py create_dataset local
```

Once the code has finished running, you will see output in the path given to the argument $final\_output\_path$ in the configuration file. See the above section for

details about the output file structure.

A.2.2 AWS Mode

To run BeatDB v3 on AWS, a few prerequisites need to be met on the AWS platform. The configuration file contains AWS specific arguments, detailing which SQS queues to use, which S3 buckets to read from, which folders in the bucket to read from and write to, among other arguments. A file structure for the S3 bucket is shown in the following file structure diagram.



The SQS queues specified in the configuration file must exist in SQS. The S3 bucket described in the configuration file must contain a folder named *data/*. This folder must contain subfolders named for each signal type (in lowercase) and the subfolders used for the *intermediate_file_path* described in the config file (for example, the *data/* folder in the S3 bucket would need to contain *abp/*, *abp_npy/*, *eeg/*, and *eeg_npy/* subfolders, assuming the configuration file in section A.5 is used and only ABP and EEG signal types are implemented). The **_npy/* folders are used to store chunk files containing the output from population/extraction runs. The S3 bucket must also have a *bin/* folder containing all runnable code for BeatDB in the file structure described

in section A.1. Final dataset output from BeatDB's condition detection/aggregation stage will be stored in the *beatdb_output/*. Logs will be uploaded to *logs/* once worker and master instances have completed the run.

To run BeatDB v3's stage in AWS mode, run the following commands in the terminal (if your configuration file is not named $config.yaml$, you will need to also supply a $--config$ argument with the name of your config file, including the file extension):

```
python main.py population_extraction master
python main.py condition_detection_and_aggregation master
```

The first command calls the Master instance for the population/extraction module, which reads the configuration file and initializes required objects for the pipeline. It starts by uploading its configuration file to the *bin/* folder in the S3 bucket to ensure that the same configuration file is used in all instances. The Master instance creates all of the requested Worker instances, supplying them with a bootstrap script that downloads the *bin/* folder from the S3 bucket and runs the worker code. The Master instance then creates messages for all files in the S3 bucket's $data/*signal\_type*$ and adds them to the SQS queue. Once this is done, it waits until the queue is empty and all worker messages have completed processing their existing messages and writes its logs to the S3 bucket before ending the process.

Once the worker instances have downloaded the BeatDB v3 code from the *bin/* folder, they launch the worker process, which pulls messages from the queue and processes the files described by them. For population/extraction, this means each worker parses the file described by the message into chunks and then extracts features from those chunks. These chunks are written to a .npy file, like normal, but in AWS mode, this .npy file is written to the $data/*signal\_type*\_npy$ folder in the S3 bucket. After the file has been uploaded, it is deleted locally while the worker instance grabs another message from the queue. Worker instances survive until there are no more messages on the queue. If a worker attempts to pull a message from an empty queue, it will either stop or be terminated, depending on the $ec2\_stop$ argument in the configura-

tion file.

After the first command has completed its run, the second command can be run. As with population/extraction, this stage launches a master instance that has the same general logic. The difference between these stages are the input and output paths in the S3 bucket and how the messages are processed in the worker instances. For this stage, the S3 input path is the output path of the previous stage ($data/*signal\_type*\_npy$) and the S3 output path is $beatdb\_output/$. This means that the messages in the SQS queue contain links to these .npy files instead of the original raw data files. The worker instances are prepared for this and process the chunks using the rolling window condition detection algorithm and feature aggregation. Once the resultant dataset has been generated for the file, it is written to $beatdb\_output/$ and then deleted locally. Other than these differences, the master/-worker logic is the same between both stages of BeatDB v3 in AWS mode, and is reflected in the codebase (since Master and Worker classes are superclasses of specific stage Master and Worker classes).

A.3 Implementing Multi-Channel and Multi-Sample Feature Extraction

BeatDB v3 allows researchers to compute multi-channel and multi-sample features from specific types of physiological data. For a definition of these terms, please refer to section 4.2.3. This section will detail the implementation specifics regarding multi-channel and multi-sample feature extraction, walking through the implementation present in the EEG signal type as an example of extraction of these advanced types of features.

At the time of this writing, only the EEG signal type has the implementation for multi-channel and multi-sample features directly built in. As with other signal types, the population/extraction stage will call the edf_to_chunks function from the signal

type only, while the function is free to call any of the other functions available to it, allowing signal types to be flexible in the way that they process their chunks.

For the EEG signal type, the *edf_to_chunks* function parses the data from the raw file into a Python friendly form. The EEG signal type data is represented as a sequence of samples, with each sample containing 11 readings from a specific point in time. The EEG sample data is arranged in an array of length 11, as seen in Table A.1 (descriptions of data columns from Motion Artifact Contaminated fNIRS and EEG Data page on PhysioNet[1]). Once this data is available in a multi-dimensional Python array, the *edf_to_chunks* function passes this data to the *extract_features_from_samples* function, specific to the EEG signal type. This function is where the computation specific to multi-channel and multi-sample features is located.

Index	Value type
0	Sample index
1	Channel 1 : Raw EEG : sampled @ 2048 Hz
2	Channel 2 : Raw EEG : sampled @ 2048 Hz
3	Trigger data for EEG data : sampled @ 2048 Hz
4	Accelerometer 1 : X-axis : sampled @ 200 Hz
5	Accelerometer 1 : Y-axis : sampled @ 200 Hz
6	Accelerometer 1 : Z-axis : sampled @ 200 Hz
7	Accelerometer 2 : X-axis : sampled @ 200 Hz
8	Accelerometer 2 : Y-axis : sampled @ 200 Hz
9	Accelerometer 2 : Z-axis : sampled @ 200 Hz
10	Trigger data for accelerometer data : sampled @ 200 Hz

Table A.1: Layout of data channels contained in a single sample of data from the EEG dataset

The layout of the code in *extract_features_from_samples* is relatively straightforward but relies on a few tiny details throughout the system. The function takes in the parsed signal data along with an argument from the config file that specifies the length of the rolling window used to compute multi-sample features, called *rolling_window_length*. The function then creates an array in length equal to this

[1] https://physionet.org/pn4/motion-artifact/

argument called *collected_samples*, which stores the last *rolling_window_length* samples seen. For multi-sample features, the process is relatively simple. Once the *collected_samples* array has been filled with *rolling_window_length* samples, multi-sample features will be computed using functions specified in the configuration file under the *rolling_window_features* argument on the chunks in the *collected_samples* array. This process is almost exactly similar to the aggregation process in the condition detection/aggregation stage. Once the multi-sample features have been computed for a specific sample, it is added to the *collected_samples* array while the oldest entry in the array is removed, so that the *collected_samples* array maintains a length of *rolling_window_length*. Keep in mind that this process means that the first *rolling_window_length* samples in a raw data file will not have values for the multi-sample features, since the *collected_samples* array did not yet have a length of *rolling_window_length* for these samples.

The computation of multi-channel features is even simpler than the computation of multi-sample features. For multi-channel features, the only thing that needs to be programmed is the multi-channel feature functions themselves. For single-channel signal types, feature functions typically take in a list of samples that represent a beat and use these values to compute some feature value. In a multi-channel signal type, such as the EEG signal type, the feature functions take in a list of data channel values for a single sample. Since each feature function gets the list of data channel values for a single sample, the feature functions can parse the data with the knowledge of what each value represents and compute the multi-channel features accordingly. This is made simple by the nature of the feature functions, which normally return a single value but can also return a dictionary of key-value pairs that will be added as feature values for the sample after calculation.

This simple paradigm of allowing the feature functions to return both dictionaries and single values for feature computation allows multi-channel feature computation to be simple and flexible to the signal types needs, whether that be computing additional features for each individual sample, as is done currently in the EEG signal

type, or computing features from chunks. Shifting to this is rather simple, since the only thing that this change of input would require is that the implementations of each feature function in a specific signal type are consistent with the nature of their inputs and outputs. For example, to shift the EEG signal type from computing features for each individual sample to computing features for a group of samples (a chunk), the user simply needs to write the feature functions with the assumption that the feature functions will take in a list of list of data channel values (representing multiple samples) rather than a list of data channel values (representing a single sample).

This design pattern contains the computational messiness within the signal type implementation, since the functions that parse the feature function return values into a dictionary of all feature values only care about the final values computed from each feature function, and whether the functions return single feature values or multiple feature values. This allows multi-channel and multi-sample features to be easily calculated within any implementable signal type without the need to rewrite BeatDB source code.

A.4 BeatDB v3 Pitfalls

If you are performing an experiment with BeatDB v3, you should probably make sure that lag is at least 3 times as large as the number of subwindows you want to split the lag window into. This is to ensure that each subwindow has at least three beats in it, so that the aggregated features can be more useful information than the aggregation function base cases of two values.

If you are adding code to BeatDB v3, please make sure that you really think about what the code is doing. The most costly operations are those that iterate over the data, so when implementing a new method, beward that your code does not increase passes over the data unless absolutely necessary. It is easy to introduce additional loops over the data if not careful. For example, the util function for finding beat index from time would be significantly slower if a starting index wasn't sup-

plied as an argument. Even then, calculating all three indices using the function *get_groupwindow_beat_indices_from_time* proved to be even more efficient than calling *get_beat_index_from_time* three times, so always be sure to think about how you're going about the implementation of a new feature.

Make sure that the *bin/* folder in the S3 bucket is the exact same code that is being used to run the Master instance! If the *bin/* folder is not up to date with your local changes when you run the commands, your output will not be what you expect and you'll probably be confused for a while until you figure out that they weren't synced. This can easily be done with the s3 cli provided by Amazon.[2]

[2] https://docs.aws.amazon.com/cli/latest/reference/s3/sync.html#examples

A.5 BeatDB v3 Configuration File

config.yaml

```
-----------------------------------------------------------------


aws:
    dynamo_table_prefix:    beatdb_v3_test
    s3_bucket:              beatdb-v3
    s3_bin_folder:          bin
    ec2_image_id:           ami-16e4d001
    ec2_key_name:           aws_alfacsail_baldo
    ec2_instance_profile:   beatdb_role
    ec2_num_instances:      1
    ec2_instance_type:      r3.large
    ec2_stop:               yes
    sqs_visibility_timeout: 3600
    initialize_tables:      true


preprocessing:
    s3_raw_folder:          data/raw
    s3_output_folder:       data/preprocessed
    sqs_queue_name:         beatdb_v3_preprocess
    preprocessing_scripts:
        - scripts/preprocessing/butter_lowpass_filter.py


beatdb:
    final_output_path:      beatdb_output/
    # used to store .npy files from population_extraction
    # if run in standalone
    intermediate_file_path: beatdb_output/abp_npy/
```

```
    sampling_frequency:     125  # in Hz (sec^-1)
    signal_type:            abp

population_extraction:
    gap_beat_cutoff_length: 6  # in seconds; must be >= 3 seconds
    data_path:              data/abp/
    rolling_window_length:  100  # in number of beats/chunks
    features:
        - crest
        - diastole
        - diastole_duration
        - kurtosis
        - mean_arterial_pressure
        - mean
        - pressure_area
        - pulse
        - rms
        - skew
        - std
        - systole
        - systole_duration

condition_detection_and_aggregation:
    # condition detection
    lead:                   60  # in seconds
    lag:                    30  # in seconds
    condition_window_len:   1800  # in seconds
    # condition_function params automatically set to 90%
    # (percentage) of beats with MAP < 60 mmHg (threshold)
    condition_function:     ahe
```

```
# aggregation
subwindows:             10
aggregators:
    - kurtosis
    - mean
    - skew
    - std
    - trend
# list of features from population_extraction to NOT aggregate
excluded_features:
    -
```

Appendix B

Algorithms

Algorithm 1 Rolling window pseudocode used in condition detection/aggregation stage in BeatDB v3

```
1: procedure detect_conditions(list_of_list_of_chunks)
2:     for chunks in list_of_list_of_chunks do        ▷ Analyze one file at a time
3:         windows = []
4:         while start_ind < len(chunks) do
5:             find lead_start_index, condition_window_start_index, and window_end_index
6:             if gap beat in group window range then
7:                 start_ind = first valid beat after gap beat
8:                 jump to start of while loop
9:             end if
10:            subwindows = find_subwindows(chunks)
11:            classification = user_def_condition_function(condition_window)
12:            windows.append(window object created from bound indices, subwindows, and classification)
13:            if classification == False then
14:                //start_ind+ = lag                ▷ v1 style jumps, commented out
15:                start_ind+ = len(chunks) ∗ .10      ▷ v2 style jumps
16:            else
17:                start_ind = window_end_index + 1    ▷ Don't overlap windows
18:            end if
19:        end while
20:        write_windows_and_subwindows_to_files(windows)
21:    end for
22: end procedure
```

Algorithm 2 Find subwindows pseudocode used in condition detection/aggregation stage in BeatDB v3. This code is simplified from the actual implementation for readability but is very similar in process.

```
1: procedure find_subwindows(all_file_chunks)
2:     subwindows = []
3:     all_subwindows_chunks = split_into_even_groups(all_file_chunks)
4:     for subwindow_chunks in all_subwindows_chunks do
5:         agg_features = aggregate_all_features(subwindow_chunks)
6:         create subwindow object with index bounds and agg_features
7:         subwindows.append(subwindow)
8:     end for
9:     return subwindows
10: end procedure
```

Algorithm 3 Feature aggregation pseudocode used in condition detection/aggregation stage in BeatDB v3. This code is simplified from the actual implementation for readability but is very similar in process.

```
1: procedure aggregate_all_features(subwindow_chunks)
2:     agg_features = Map()
3:     for agg_function in aggregation_functions do
4:         feature_values = all values of a specific feature from subwindow_chunks
5:         agg_feature_name = feature_name + "_" + agg_function_name
6:         agg_features[agg_feature_name] = agg_function(feature_values)
7:     end for
8:     return agg_features
9: end procedure
```

Bibliography

[1] M. Akin. Comparison of Wavelet Transform and FFT Methods in the Analysis of EEG Signals. *Journal of Medical Systems*, 26(3):241–247, 2002.

[2] Amazon. DynamoDB Local. Version 1.0. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/DynamoDBLocal.html. Accessed 10/26/2016.

[3] A. Baldominos. *BeatDB v2 Documentation*. Anyscale Learning for All, CSAIL, 2016.

[4] A. Baldominos and ALFA. PhysioMiner System Architecture. Meeting with Philips, 2016.

[5] M. C. de Jongh, A. C. Maan, E. T. van der Velde, and C. A. Swenne. A Wavelet-Based Artifact Reduction From Scalp EEG for Epileptic Seizure Detection. *IEEE Journal of Biomedical and Health Informatics*, 2015. http://ieeexplore.ieee.org/document/7158988/. Accessed 03/24/2017.

[6] M. C. de Jongh, A. C. Maan, E. T. van der Velde, and C. A. Swenne. Serial ECG analysis after myocardial infarction: When heart failure develops, the ECG becomes increasingly discordant. *Computing in Cardiology Conference (CinC)*, 2016. http://ieeexplore.ieee.org/document/7868778/. Accessed 04/01/2017.

[7] F. Dernoncourt. BeatDB: An end-to-end approach to unveil saliencies from massive signal data sets. Master's project, Massachusetts Institute of Technology, CSAIL, Feb. 2015. http://alfagroup.csail.mit.edu/12.x/tiki-download_file.php?fileId=144&display. Accessed 09/26/2016.

[8] F. Dernoncourt, K. Veeramachaneni, and U.-M. O'Reilly. BeatDB: A large scale waveform feature repository. *NIPS 2013, Machine Learning for Clinical Data Analysis and Healthcare Workshop*, page 4, Dec. 2013. http://alfagroup.csail.mit.edu/12.x/tiki-download_file.php?fileId=135&display. Accessed 09/26/2016.

[9] M. Douglass, G. Clifford, A. T. Reisner, G. B. Moody, et al. Computer-assisted de-identification of free text in the MIMIC II database. *Computers in Cardiology, 2004*, 2004. http://ieeexplore.ieee.org/document/1442942/. Accessed 04/10/2017.

[10] V. Gopal. PhysioMiner: A scalable cloud based framework for physiological waveform mining. Master's project, Massachusetts Institute of Technology, CSAIL, June 2014. http://alfagroup.csail.mit.edu/12.x/tiki-download_file.php?fileId=152&display. Accessed 09/26/2016.

[11] D. Hanley, L. S. Prichep, J. Bazarian, J. S. Huff, et al. Emergency department triage of traumatic head injury using a brain electrical activity biomarker: A multisite prospective observational validation trial. *Academic Emergency Medicine*, 2017. https://www.ncbi.nlm.nih.gov/pubmed/28177169. Accessed 04/06/2017.

[12] N. Hazarika, J. Z. Chen, A. C. Tsoi, and A. Sergejew. Classification of EEG signals using the wavelet transform. *Digital Signal Processing Proceedings*, 1997. http://ieeexplore.ieee.org/document/627975/. Accessed 03/24/2017.

[13] P.-W. Huang, S.-C. Tang, Y.-M. Lin, Y.-C. Liu, et al. Predicting stroke outcomes based on multi-modal analysis of physiological signals. *Digital Signal Processing (DSP)*, 2015. http://ieeexplore.ieee.org/document/7251913/. Accessed 04/01/2017.

[14] M. Längkvist, L. Karlsson, and A. Loutfi. Sleep stage classification using unsupervised feature learning. *Advances in Artificial Neural Systems*, 2012.

[15] H. Lee, C. Ekanadham, and A. Y. Ng. Sparse deep belief net model for visual area v2. In J. C. Platt, D. Koller, Y. Singer, and S. T. Roweis, editors, *Advances in Neural Information Processing Systems 20*, pages 873–880. Curran Associates, Inc., 2008.

[16] J. Lee, D. J. Scott, M. Villarroel, G. D. Clifford, et al. Open-access MIMIC-II database for intensive care research. *Engineering in Medicine and Biology Society, EMBC, 2011 Annual International Conference of the IEEE*, 2011. http://ieeexplore.ieee.org/document/6092050/. Accessed 04/06/2017.

[17] Q. Li, R. G. Mark, G. D. Clifford, et al. Artificial arterial blood pressure artifact models and an evaluation of a robust blood pressure and heart rate estimator. *Biomedical Engineering Online*, 8(13), 2009.

[18] B. H. McGhee and E. J. Bridges. Monitoring arterial blood pressure: what you may not know. *Critical Care Nurse*, 22(2):66–79, 2002.

[19] Y.-J. Min, H.-K. Kim, Y.-R. Kang, G.-S. Kim, et al. Design of Wavelet-Based ECG Detector for Implantable Cardiac Pacemakers. *IEEE Transactions on Biomedical Circuits and Systems*, 2013. http://ieeexplore.ieee.org/document/6471259/. Accessed 04/11/2017.

[20] G. B. Moody, R. G. Mark, and A. L. Goldberger. PhysioNet: a Web-based resource for the study of physiologic signals. *IEEE Engineering in Medicine and Biology Magazine*, 20(3):70–75, 2002. http://ieeexplore.ieee.org/document/932728/. Accessed 04/06/2017.

[21] W. Ren, M. Han, J. Wang, D. Wang, et al. Efficient feature extraction framework for EEG signals classification. *Intelligent Control and Information Processing (ICICIP)*, 2016. http://ieeexplore.ieee.org/document/7885895/. Accessed 04/01/2017.

[22] M. Saeed, M. Vilarroel, A. T. Reisner, G. Clifford, et al. Multiparameter intelligent monitoring in intensive care II (MIMIC-II): A public-access ICU database. *Critical Care Medicine*, 39(5):952–960, 2011.

[23] K. A. Salam and G. Srilakshmi. An algorithm for ECG analysis of arrhythmia detection. *Electrical, Computer and Communication Technologies (ICECCT)*, 2015. http://ieeexplore.ieee.org/document/7226130/. Accessed 04/01/2017.

[24] C. Spencer, J. Pickhardt, N. Gauthier, N. Carroll, et al. fake-s3. Version 0.2.4. https://github.com/jubos/fake-s3. Accessed 10/26/2016.

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

[26] K. T. Sweeney, H. Ayaz, T. Ward, M. Izzetoglu, et al. A Methodology for Validating Artifact Removal Techniques for Physiological Signals. *IEEE Transactions on Information Technology in Biomedicine*, 16(5):918–926, 2012. http://ieeexplore.ieee.org/document/6236173/. Accessed 03/20/2017.

[27] K. T. Sweeney, S. F. McLoone, and T. Ward. The Use of Ensemble Empirical Mode Decomposition With Canonical Correlation Analysis as a Novel Artifact Removal Technique. *IEEE Transactions on Biomedical Engineering*, 60(1):97–105, 2012. http://ieeexplore.ieee.org/document/6332491/. Accessed 03/20/2017.

[28] Z. Syed, J. Guttag, and C. Stultz. Clustering and symbolic analysis of cardiovascular signals: Discovery and visualization of medically relevant patterns in long-term data using limited prior knowledge. *EURASIP Journal on Advances in Signal Processing*, 2007. http://dspace.mit.edu/handle/1721.1/69825. Accessed 04/02/2017.

[29] A. Szczepanski and K. Saeed. Real-Time ECG Signal Feature Extraction for the Proposition of Abnormal Beat Detection - Periodical Signal Extraction. *Biometrics and Kansei Engineering (ICBAKE)*, 2013. http://ieeexplore.ieee.org/document/6603513/. Accessed 04/06/2017.

[30] T. Tabassum and M. Islam. An approach of cardiac disease prediction by analyzing ECG signal. *Electrical, Computer and Communication Technologies (ICECCT)*, 2016. http://ieeexplore.ieee.org/document/7873093/. Accessed 04/01/2017.

[31] A. Waldin. Learning blood pressure behavior from large blood pressure waveform repositories and building predictive models. Master's project, Massachusetts Institute of Technology, CSAIL, June 2013. http://alfagroup.csail.mit.edu/12.x/tiki-download_file.php?fileId=148&display. Accessed 09/26/2016.

[32] D. Wang and Y. Shang. Modeling Physiological Data with Deep Belief Networks. *International journal of information and education technology (IJIET)*, 3(5):"505–511", 2013. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4142685/. Accessed 04/03/2017.

[33] A. Warski et al. elasticmq. Version 0.9.3. https://github.com/adamw/elasticmq. Accessed 09/16/2016.

[34] K. C. Wee and M. S. M. Zahid. Cloud Computing for ECG Analysis Using MapReduce. *Advanced Computer Science Applications and Technologies (ACSAT)*, 2015. http://ieeexplore.ieee.org/document/7478728/. Accessed 04/01/2017.

[35] P. Zhang, J. Liu, X. Wu, X. Liu, et al. A novel feature extraction method for signal quality assessment of arterial blood pressure for monitoring cerebral autoregulation. *Bioinformatics and Biomedical Engineering (iCBBE)*, 2010. http://ieeexplore.ieee.org/document/5515739/. Accessed 04/09/2017.

[36] W. Zong, T. J. Heldt, G. B. Moody, and R. G. Mark. An open-source algorithm to detect onset of arterial blood pressure pulses. *Computers in Cardiology, 2003*, pages 259–262, 2003. http://ieeexplore.ieee.org/document/1291140/. Accessed 04/06/2017.