Software Design Document

GNomes

Members

Jacob Christiansen

Allen Clarke

Yuanyuan Fu

John Jackson

Mentor

Mahsa Keshavarz

Clients

Dr. Toby Hocking

Christopher Coffey

Version 2.1

February 14, 2020

Contents

1 Introduction

We have very little knowledge of our own genome. Many diseases are caused by mutations in our DNA, but we have no idea how to alleviate symptoms, much less cure or prevent these diseases. If there were a way to determine what genetic diseases (like cancer) someone was susceptible to before the disease has progressed far enough to show symptoms, we could effectively prevent these diseases well before they are life-threatening. This could be possible through the study of our own genome.

The genomics industry is so new and cutting-edge that there are not very many developed tools to help scientists analyze genomic data and understand these diseases. To learn more about genetic diseases, a massive amount of data would have to be analyzed by biologists, charting which genes are active in patients with diseases by hand and establishing correlations between sick patients and their DNA. Those trends would have to be compared to the trends in the DNA of healthy patients to look for more trends. This is a very tedious process and distracts biologists from more creative research, since they are spending days looking over genomic data and writing down trends they see by hand instead of staying on the cutting edge of research and trying to come up with ways to cure or reduce the trends they are seeing. This is where Dr. Toby Hocking is involved.

Dr. Hocking is an assistant professor at NAU and is also the Machine Learning Lab Director. He works with other researchers to develop machine learning algorithms for other fields of study. One of his main fields of interest is genomic research data. Dr. Hocking has already completed multiple research projects looking at DNA data and coming up with algorithms to graph the data in a similar way to how scientists do so now. The goal is to use supervised machine learning to look at genomic data and to bring scientists and statisticians together.

Dr. Hocking wants to bridge the gap between biologists and statisticians because nobody else has done it yet, and doing so will improve the field of work for both parties. This will help statisticians because they have the tools to write complex models to analyze data but are missing new sets of data to look at. Receiving data from biologists will open a new door for statisticians and allow them to analyze more data as well as provide useful insight to scientists. Having new analysis tools will help biologists analyze much larger pieces of data much more efficiently, after having put in some labels about the data for the statisticians to use as a key to compare models against. Overall, the sharing of genomic data is clearly beneficial in uniting both parties' investigations into genomic diseases.

Team GNomes is working with Dr. Toby Hocking to create PeakLearner, a machine learning web app to process genomic data. This new tool will:

- help scientists save time combing through data and allow them to get back to making new medical research breakthroughs
- improve on current unsupervised peak-detection algorithms by reducing false positives and false negatives
- allow for greater research collaboration between machine learning specialists and genomic researchers

Peak Learner will fulfill these key **functional requirements**:

- Users can upload their genomic coverage sets to a publicly accessible website and view them in an embedded genome browser.
- Users can manually add one of four labels—Peak, NoPeak, PeakStart, and PeakEnd—to mark peak information in their data sets, and users can delete or modify these labels.
- The server, upon receiving the label and coverage data from the client, retrieves the closest-matching model from the database. If that model is not within an acceptable accuracy, the Monsoon cluster calculates and returns a new model.
- The server returns a model, which renders on top of the user's coverage data.
- Users can download models.

We specify that PeakLearner meet these **performance requirements**:

- *Speed.* The latency between a user making a label and seeing an updated model should not be greater than 0.2 seconds.
- *Simplicity.* Creating or editing a label should not require more than two clicks.
- *Accuracy.* The returned peak model should not contradict any of the user's labels.

Finally, we must accept these **environmental requirements**:

- Monsoon uses the SLURM job-scheduling system.
- Dr. Hocking's GFPOP algorithm is written in R.
- Users will login using Google authentication.
- PeakLearner's source code and documentation will be publicly available on GitHub.
- PeakLearner will use the Travis-CI continuous integration system.

2 Implementation Overview

Now that PeakLearner has been introduced and its requirements stated, this section will examine the key technologies involved in our implementation. PeakLearner is a web application that uses supervised machine learning to detect abnormal peaks in genomic data supplied by the user. It consists of 4 components:

1. the web interface, consisting of an embedded genome browser and a text box for submitting URLs;
2. the web server, a backend and central controller of the system, sending and receiving data from the database and the cluster;
3. the database, which stores peak models and error matrices; and
4. the GPU cluster, where new models are calculated.

2.1 Key Technologies

What follows are the languages and tools we will use in our implementation.

2.1.1 Web interface

- The front-end is coded with standard HTML, CSS, and JS.
- Retrieves BigWig file from a URL supplied by the user.
- JBrowse, the genome browser we will be modifying, is embedded on the page in a div. JBrowse is the graphical tool that allows users to view their data, create labels, and view peak models.

- JBrowse is coded in JS, using the Dojo framework.
- Two JBrowse plugins that are important for this project are Wiggle-Highlighter and MultiBigWig. WiggleHighlighter provides the labeling mechanism and MultiBigWig lets us view multiple bigWigs on the same track.
- The user creates labels on their data in the JBrowse window. Four values—chromosome number, chromosome start and end, and label type—are sent to the server in a JSON object.
- Models retrieved from the database or generated by the cluster are displayed on the browser as a BigWig overlaid on the original data.

2.1.2 Server

- The server is coded in Python. We're using Python because its NumPy library is convenient for handling matrices.
- The server interfaces with the database using Python's bsddb3 module.
- The server encodes retrieved models as JSON objects and sends them to the web browser.

2.1.3 Database

- The database implements the Berkeley DB software library. We're using Berkeley DB because its a lightweight NoSQL that will help PeakLearner achieve optimal speed.

2.1.4 GPU cluster

- The Monsoon cluster uses the SLURM Workload Manager to execute jobs.
- Dr. Hocking's GFPOP algorithm, written in R, runs on the cluster along with the user's coverage data and labels to calculate a new model.

3 Architecture Overview

Now that we have looked over the basic implementation of the system, we will now take a look at the architecture of the system. First, we will look at the overview of the entire system and spend the rest of this section going into detail about how each of the components communicates with each other.

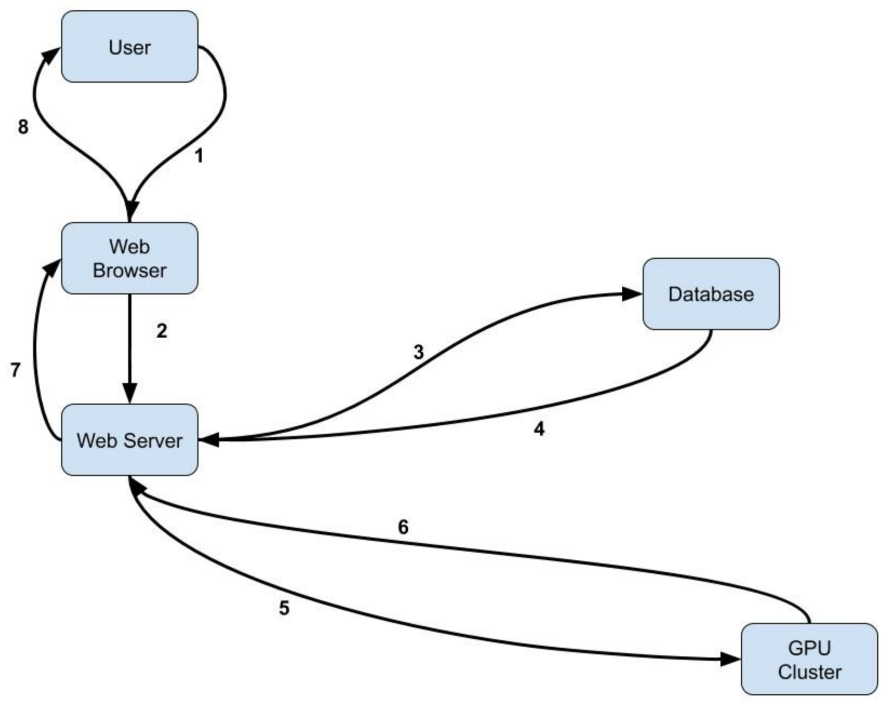

Figure 1: This is the basic outline of each component and how components communicate with each other.

3.1 Discussion

Figure 1 shows the key components of our system:

- Web browser – This is the front-end of the system. Users will load their data into JBrowse on the front-end and interact with the web browser to add and modify labels on their genomic data. JBrowse has many sub-pieces, like WiggleHighlighter and MultiBigWig, which are used so a user properly see and interact with the data.

- Web server – This is the central controller of the system. The server will act as a middleman to pass information from the browser to the database and back again. The server will also pass data along to the GPU Cluster when the system needs new models to be calculated.

- Database – The database is used to store past models and labels that were created on the web server. The models displayed on the web browser are stored in the database and passed to the browser through the web server.

- GPU cluster – The GPU cluster (Monsoon) is used to calculate new models for the database. When the server finds no optimal models in the database, it sends information to the cluster so it can calculate new models.

3.1.1 Module interactions

In **Figure 1** above, every arrow is numbered to accurately describe the interactions between each major component of the system.

1. Users interact with the web browser by uploading their data. This can be done by providing a file or a URL to a file. Users can also click and drag over their data in the JBrowse genome browser to add labels to their data. These labels can denote one of four things: Peak, NoPeak, PeakStart, or PeakEnd. These labels are used to generate models and are passed on to the web server in step 2.

2. Following that these labels along, the information about the labels is sent from the web browser to the web server. This will be done using something like AJAX to send data to the server using GET and POST requests. The purpose of this step is sending along the information users input to the server so that it can be processed accordingly and an accurate model can be found/created.

3. After receiving data, it is the server's job to pass models and labels into the database. The database is then responsible for storing the model and using labels to find the most accurate model in the database that fits the given labels.

4. Once the database has received the data, it must use the labels entered by the user and provided by the server to send the optimal model back to the server. If there is no optimal model, then the server triggers steps 5 and 6. If there is a model returned, the server skips to step 8.

5. If the DATABASE does not have a model to display that accurately fits the labels, then the SERVER sends the labels and data off to the GPU CLUSTER. The CLUSTER needs to take the data and labels and calculate a new model for the user to see that accurately fits the labels.

6. Once the GPU CLUSTER has calculated a new model, it sends the model back to the SERVER. This new model was calculated by the user inputted labels that were given to the SERVER in step 2. After this model is calculated, it is also sent to the DATABASE to be stored and retrieved in future model calculations, which is a sub-function of step 3.

7. At this point, the SERVER has received a model that is fit for display to the user (from either step 4 or 6). This model is then encoded and sent back to the WEB BROWSER as a response to the POST or GET request sent in step 2.

8. Once the WEB BROWSER has gotten a model from the SERVER, it is decoded and then displayed in the embedded JBrowse window. This is the means of getting a new and updated model back to the user so they can view a model that fits all of their given labels.

4 Module and Interface Descriptions

Now that the general architecture has been introduced, this section will cover the project's modules in more thorough detail. We use "module" to mean a functional component of the overall project. It could be something like a GUI for an app or a set of scripts that automate a process. In essence, a module is a cohesive group of objects in a software engineering project that work together towards a goal.

To explain our project, it's necessary to introduce some terminology that's specific to the field of machine learning. This project uses a specific style of machine learning called "supervised learning." Supervised learning entails having a set of tests called the training data, and answers called labels, by which a program modifies itself through repeated iterations. An example of this is spam filtering; the training data would be a set of emails, and the labels would be the categories "spam" and "not spam."

With an understanding of the terminology, this section will go over each module of PeakLearner. To start, there are two main sections between the client-side and the server-side. On the front-end is the genome browser we are implementing called JBrowse. On the back-end there are the server, the database, and the compute cluster.

4.1 Front-end

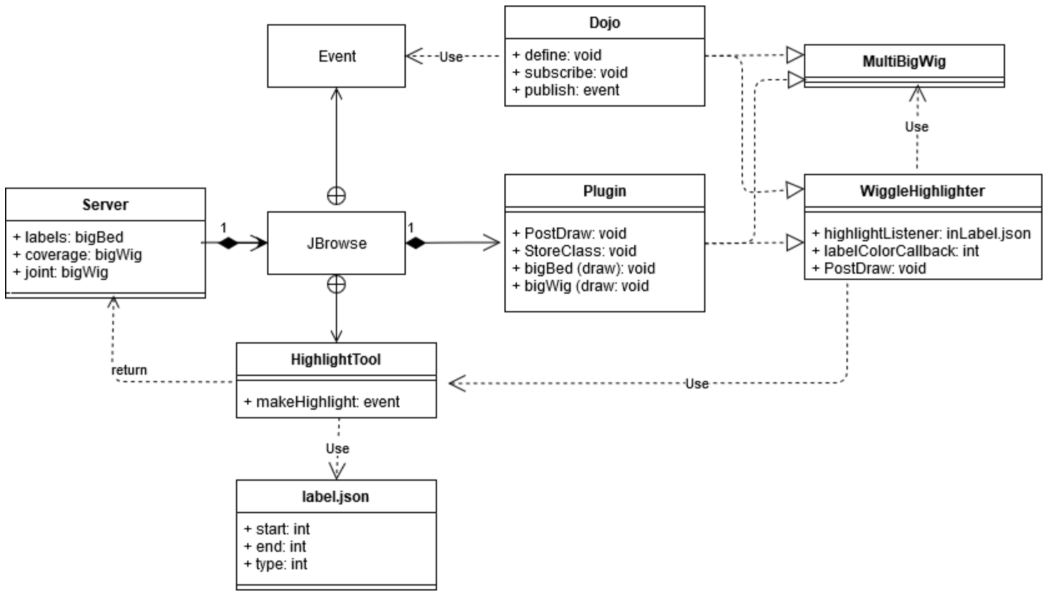

Figure 2: A UML of the front-end.

One of the most important components of the system is the genome browser we are extending called JBrowse. Shown in **Figure 2** is a diagram illustrating the elements of JBrowse that are salient to this project. JBrowse has four very important jobs, as it is the only point of contact between the rest of the system and the user. Its jobs include taking in the files the user wants to display, displaying the data, letting the user create and edit label data, and finally passing the appropriate data off to the back-end server.

Firstly, we must be able to take in the data the user wants to display. This means having the user pass in a URL to the bigWig data they want to show and analyze. To accomplish this, the plan is to have an input box where the user can either pass in a URL to a specific file to display a track hub, which is a collection of many files. The track hub option will allow many tracks to be generated at once rather than sequentially. Once the user inputs the URL, a script will send the tracks configuration file with the URL and all other information in order to appropriately show the data to the user. Some of this information would be two other files that the back-end server needs to create. The first file is the model we will be computing, and the second is the empty bigBed file to store the labels the user will input. The model will be generated by an algorithm given to us by Dr Hocking. To summarize, the user will input a URL containing the information they want to analyze. This causes JBrowse to ask the back end server for two more files, the first the first model based off of the URL data, and second an empty file to hold the users labels. Finally, when JBrowse gets all of this information together, it appends to a configuration file to create the track that will show the information the user input and the first guess model.

Secondly, the browser acts as a display. Through the help of JBrowse, the browser is able to display the BigWig file in a way that is visually easier to interpret and than the original file format. The JBrowse display is showing multiple things simultaneously, which include: the user's original data provided from the URL, an optimal model that best fits the data and labels, and labels added by the user to modify a model. This graphic representation of all three pieces is very important to the system, since without a visual display to clearly see peaks, there is no point in having users add labels to modify the data since they wouldn't be able to see their updated results. To do this we are using two plugins for JBrowse, MultiBigWig and WiggleHighlighter. The most important one is WiggleHighlighter, which 1) allows the labels to be shown over the other data as they are stored as a bigBed file, and 2) extends MultiBigWig in order to show both the original data and the model on the same track as these are both bigWig files.

Thirdly, the browser must allow for a user to edit their data. This is done by adding labels on top of the model drawn with JBrowse. These labels will be colored rectangles drawn over the data, and each color represents a different one of the four labels (Peak, NoPeak, PeakStart, PeakEnd). By adding these labels on top of the JBrowse display, the user can take note of which labels are affecting which region of the sequencing data that is displayed. This is accomplished by creating a listener for the built-in highlight tool of JBrowse to capture when a user highlights a section and where in the genome it is. As for switching between kinds of labels we will override the onClick behavior of the track in order to be able to cycle through the four options. This information is used by the final responsibility of the browser.

Lastly, the browser must be able to send data off to our server. This is another critical feature of the system as, without it, the browser would not be able to receive updated models to show the user. In this step, the browser takes all of the user added labels and encodes them into a JSON object, which it then sends through an AJAX call to a special route built on the server designed to listen for this object. Once the user adds labels and the data is encoded and sent, the browser waits for a response to its POST request from the server, which will contain the new and improved model to display back to the user.

4.2 Back-end

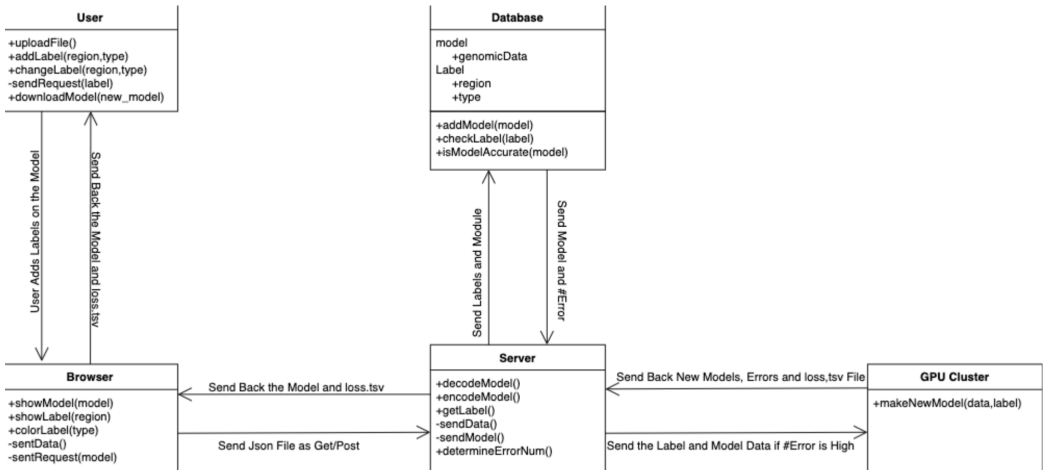

Figure 3: A UML of the back-end.

4.2.1 UML method descriptions

- User
  - uploadFile() – upload the bigWig file on the web app
  - addLabel(region, type) – add labels to the model
  - changeLabels(region, type) – change the information of the label
  - sendRequest(label) – submit the information of the label
  - downloadModel(newModel) – download the model from browser
- Browser
  - showModel(model) – show the visual display of the bigWig file after the user uploads it
  - showLabel(region) – display the region of labels on the visual model after the user adds them
  - colorLabel(type) – color the label to show the type of label
  - sendData() – send the data to the server
  - sendRequest(model) – send request to get the new model

- Server
  - decodeModel() – decode the model
  - encodeModel() – encode the model
  - getLabel() – get label information from the browser
  - sendData() – send label information to the database and GPU cluster
  - sendModel() – send model data to the database and GPU cluster
  - determineErrorNum() – determine the number of errors to decide if the data should be sent to the GPU cluster
- Database
  - addModel(model)—add the model to the database
  - checkLabel(label)—compare the labels against the model for accuracy
  - isModelAccurate(model)—determine if the model is accurate
- GPU cluster
  - makeNewModel(data label) – calculate new model for the database

4.2.2 Server

The server is the next major piece of the system, outlined in **Figure 3**. The purpose of our server is to act as a controller. The server will be written in python as it is an easy to use language that can communicate well with the web browser and can also communicate with the Database (outlined below in section 4.2.2) and with the GPU cluster (section 4.2.3). Most of the work happens server-side because the server needs to know where to send data at all times.

The first thing the server does is listen for the browser to send data. Once the browser has sent over some data in the form of a JSON object, the Python server decodes it. It then takes this decoded data, more specifically the labels from the data, and passes them off to the database. The database then returns a model and an error number. This error number represents the number of times the model contradicts the user's labels. The server now has a choice. If the error number is low enough to be below some threshold, or (based on the models in the database) we know that a more accurate model does not exist, then the server will encode the model into a JSON object and send it back to the browser to be displayed. On the other hand, if the server deems the error number too high, it will instead pass the labels and model data to a GPU cluster to calculate a new model. Once the cluster has calculated a new model, the model is sent back to the server so it can be put into the database. Simultaneously, the model is passed back to the browser by encoding it as a JSON object and sending it as a response to the data the server originally received from the browser.

Overall, the role of the server is to pass data around. It takes in data from the browser and directs it to the database. Once the database sends back a model, the model is either sent back to the server or the GPU cluster receives a copy of the labels and data to calculate a new model. After the cluster has calculated a new model, it is sent both to the user and to the database for use in future calculations. The server's role as director of traffic is clearly shown in the back-end UML by its central location and bi-directional connections with every other module in the system.

4.2.3 Database

For this project we chose to use a BerkeleyDB database. This is a non-relational database, so while it gives up some of the luxuries of a traditional database, like ease of use and readability, it has more than enough speed. This is important because we are trying to get model data to the user as quickly as possible. The reason for the speed is so that users can add labels and immediately see changes in the model they are looking at to reflect the newly added labels, encouraging them to add more labels and further improve the model. The database has three main features that it must be able to do.

First there must be a way to add a new model to the database. This new model is given to the database by the server and needs to be stored as a relevant model to compare all incoming labels against as there is a chance the new model is the most accurate model. Any new model is assumed to be stored indefinitely unless explicitly told otherwise.

Second, the system needs a way to compare the models against the labels for accuracy. In every case, either a label conforms with a given model, or it does not. For example, a label denoting NoPeak in a region of the data is either accurate or inaccurate. There is no way for a peak to only halfway in the NoPeak Label, as a peak halfway in should have either a PeakStart or PeakEnd Label instead of a NoPeak Label. After breaking down whether a label is correct or inaccurate based on a given model, we can simplify the accuracy correction down to a 2-Dimensional Array and some addition. We will define a 2-D Array, where every column denotes one of the models in the database, and every row represents one of the labels given by the user in JBrowse. Next, we will populate that array with 0's and 1's where a 1 denotes that the model in that column does not agree with the label in that row. Similarly, a 0 signifies that the model and label do agree at that specific position. Since we know that user labels are always right, we want to return the model that agrees the best with a given set of user labels. That is, we can sum up the total number of 1's in each column, and the column with the lowest total score is the most accurate model. The score represents the number of labels that are provided that do not agree with the model we are comparing the labels against.

Once we have found the best model in the system, the database must be able to send the model and its corresponding error number (score) back to the server. The server will then decide to pass the model back to the browser or to enlist the help of the GPU cluster to calculate a new, more accurate model.

4.2.4 GPU cluster

The last major component is the GPU cluster. The cluster is responsible for calculating new models for the database when there is no accurate model already in the system. The cluster is designed to share resources across many projects, and so we are only using it when the server deems that none of our models are accurate enough to return to the user.

The majority of the work has already been given to us in the form of Dr. Hocking's machine learning algorithm for calculating new models (https://github.com/tdhock/PeakSegOptimal). This algorithm takes in the coverage data of a section of the genome as well as all of the user's labels passed from the server. It then returns a model in the form of a bigWig file and a loss.tsv file, which denotes the error in the model. This is incredibly important because the cluster is the only thing capable of generating new models to add to the database, and without new models, it is very unlikely that we will always have an accurate model to display in JBrowse.

Since the GPU cluster we will be using, Monsoon, is a shared resource, it is important that we do not overload it with requests to create new models. By requirement of NAU, we will be using SLURM, a job scheduling language to send job requests to Monsoon. This will allow the GPU cluster to prioritize the generation of models against other jobs that are sent from other projects across campus. This prioritization can change based on the amount of clusters available, the size of the job we are requesting and the frequency we are asking for new models.

By using SLURM properly we should be able to get back models when needed. Ideally, a listener process will send off requests to calculate potentially accurate models before they are requested, so when the browser sends updated labels, the most accurate model is already in our database. This will reduce latency in retrieving a new model.

5 Implementation Plan

In this section, we will go over our project execution plan. Overall, we have 3 stages this semester. The first stage is to review the project and specify a new semester plan in order to maintain satisfactory progress for the rest of the semester. The second stage is our continuing work on the project, which we've broken into 11 important intermediate goals. We have two sub-teams; one is the front-end team (Jacob and John), and one is the back-end team (Allen and Yuanyuan). Each member will focus on their own part and make sure that we work on some assignments in parallel for the sake of efficiency. In the third and final stage, we will finish the project and the final deliverables.

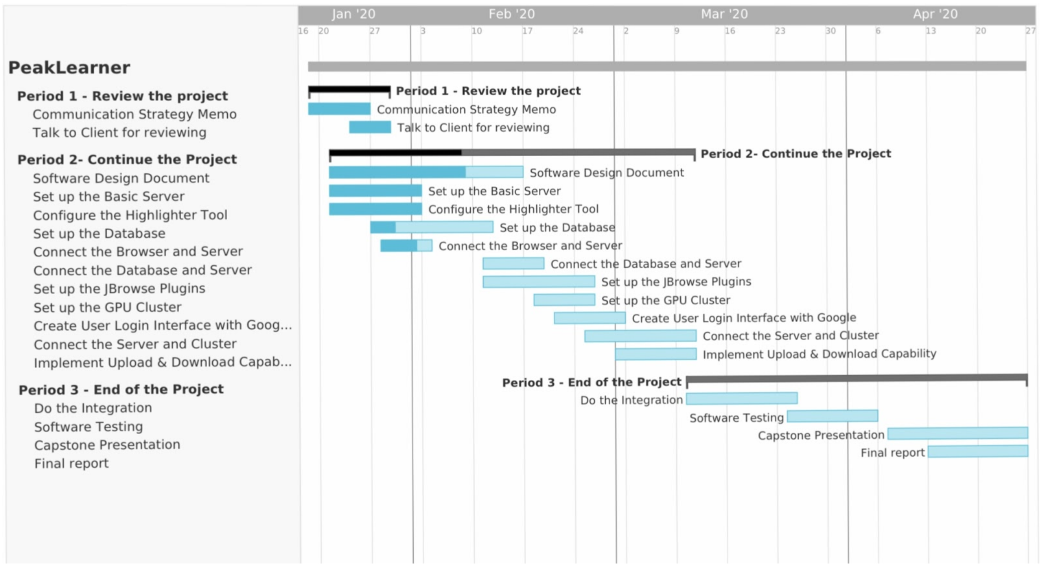

Figure 4: A Gantt chart of our implementation plan.

During last semester, we gained a thorough understanding of the project and our client's needs. In addition, we have clear functional requirements, non-functional requirements and environmental requirements. For now, we are working on setting up each module of our project and connecting them to make sure the website is basically functional. By the end of the semester, we will have done integration and final testing to make sure everything is completed and prepared for client review. Pictured in **Figure 4** is a Gantt chart showing our moving schedule.

As seen in our schedule, each item is assigned to a specific subset of the team. Next to each item is a column showing the estimated progress of each assignment. As of February 14, 2020, we have finished roughly one-third of period two. We will continue on the rest of the assignments and achieve full preparation for several presentations this semester. After spring break, we will start working on the final testing and the final report, before getting ready for the Capstone presentation.

6 Conclusion

PeakLearner will be a tool that simplifies the workflow of biologists studying genomic data. It will be able to generate peaks, predicting what areas of a sample genome are being used the most. Currently, the only way for a biologist to do this work is by hand, going through millions of genes in a chromosome and marking peaks in an Excel spreadsheet. This is a hindrance to biologists for two reasons. The first is that this is a long and time-consuming process, and the second is that it is difficult to compare these data sets between samples. PeakLearner will simplify this workflow by being interactive and using machine learning to accurately predict peaks in the data using only a small number of user-generated labels.

We are working with Dr. Toby Hocking, a researcher at NAU who looks into how machine learning can support early detection of genetic diseases. He envisions PeakLearner as a machine learning web app to help process genomic data for scientists. For Dr. Hocking, PeakLearner will support two main purposes. First, with an increase of understanding of our genome, many genetic diseases could be alleviated, cured, or caught earlier. Second, he aims to advance the field of machine learning by facilitating collaboration between genomic biologists and statisticians. There is a lot of information and many experts to learn from, but little of this information is available to train machine learning algorithms on.

In this paper, we have presented a detailed software blueprint for PeakLearner, thoroughly explaining to interested readers 1) exactly how we plan to implement our proposed project, and 2) the reasons behind our implementation decisions. Early sections give a list of languages and software packages we are using, along with a broad summary of our system's architecture and an ideal workflow. The middle part of this paper goes into more rigorous detail, explaining (with the exception of how the GFPOP algorithm itself works) exactly what the modules communicate to each other, and how PeakLearner returns an optimal peak model.

Team GNomes is pleased to report that we are making steady progress. We have enlisted the aid of Colin Diesh, a GMOD researcher and the maintainer of two crucial JBrowse plugins, WiggleHighter and MultiBigWig. His modifications have already yielded useful results, as we can now highlight sections of coverage data, generating console output that would go into a JSON passed to the server. Our server team is busy writing Python scripts that will handle JSON objects passed from the browser. Members are actively researching headless browser testing and continuous integration. Overall, we are confident that we will continue to meet developmental milestones on the way to a viable product.

PeakLearner is a tool that will simplify the workflow of biologists studying genomic data. In so doing, it will help further our understanding of our genome and help develop new gene therapies. Through our meetings with our client and our hard work as a team, the project that Dr. Hocking envisioned is already taking shape. As the Gantt chart in section 5 shows, we have a plan for the rest of the project to make sure nothing is forgotten and we do not fall behind. Overall we are confident that, with this information we have compiled, we will be able to create a product that allows biologists to have a greater understanding of our own genome.