Final Report

GNomes

Members
Jacob Christiansen
Allen Clarke
Yuanyuan Fu
John Jackson

Mentor
Mahsa Keshavarz

Clients
Dr. Toby Hocking
Christopher Coffey

Version 1.0
May 7, 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 establish further trends. This is a very tedious process and distracts biologists from more creative research. This is where Dr. Toby Hocking sees an opportunity.

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, improving the field of work for both parties. Statisticians have the tools to write complex models to analyze data, but they need new datasets to analyze. Having new analysis tools will help biologists analyze 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 by getting biologists data into the hands of a statistician's machine learning algorithm.

2 Process Overview

Before we go into detail about our process, we will start by introducing our team members, their roles and the tools we are using.

2.1 Roster

- Jacob Christiansen: Team Leader, Coder
  - Jacob was elected as the team lead for two reasons. First he has the most leadership experience, being a manager in past jobs as well as the leader of his boy scout troop for several years. In addition, Jacob's well rounded skillset makes him a prime candidate to oversee and assist the other roles.
- Allen Clarke: Architect, Coder
  - Allen has chosen to make the structural decisions about the overall framework of the code because of his experience working on server code for a year. He has a good idea of what code structure is maintainable long term and what structure is not.
- Yuanyuan Fu: Release Manager, Coder
  - Yuanyuan is responsible for managing, planning and controlling the Git repository, including testing and deploying. She is familiar with various operations of GitHub.
- John Jackson: Document Editor, Coder
  - John has adopted this role because of technical document experience and his exacting approach to editing and revision.

2.2 Roles

- Team Leader. The Team Leader is responsible for running team meetings and assigning tasks to group members. They are also responsible for making sure the team is working well together and providing initial conflict resolution among team members if needed.
- Architect. The Architect is responsible for making sure that the code is implemented as it was planned. They are also responsible for making sure the code is clean, maintainable, and consistent.

- Recorder. The Release Manager is responsible for maintaining the Git repository, ensuring the commit logs are cleaned, and configuring the build tools to prepare for the release of the product in each stage.
- Document Editor. The Document Editor is responsible for delivering finished documents. This involves line-editing and revising copy and managing section assignments.

2.3 Team Meeting Format

1. The meeting starts with each team member covering what goals they finished and what goals they were stuck on over the past week. This should be similar to a scrum meeting, with the team lead acting as scrum master.
2. As a team, we examine the goals that each team member was stuck on in the past week. We focus more on the logic of a section than on the actual code as we will do a coding workshop later in the meeting.
3. Open time managed by team leader to:
    - workshop code
    - discuss long-term plans
    - ask clarifying questions
    - review client requirements
    - redistribute work as needed
    - hold internal evaluations

2.4 Tools and Document Standards

2.4.1 Version control

We used a Git repository for this project, hosted on GitHub. Each person had a development branch of their own to work on and push their incremental changes and run small tests the branches could be merged into the master branch. The master branch would always be up to date with code that just needs final testing before ready for a real release. While it is acceptable for a members personal branch to have build errors, this code should never make it onto the master branch as it has been run and tested multiple times before being committed to the master branch.

2.4.2 Issue tracking

We used our Slack channel both to communicate outside of in-person meetings and to maintain the project. The tasks were broken up by category into the front end, back end and any general tasks that need to be done.

2.4.3 Word processing and presentations

Any documents that needed to be worked on collectively or shared were dropped into the Google Drive folder. For presentations, we used Google Slides to manage the presentation and collaborate on it as needed. For virtual presentations, we used Zoom's "share screen" and "record" features to record our individual parts, which we sent to the Team Leader who assembled the final videos.

2.4.4 Composition and review

We assigned deadlines for larger documents 48 hours prior to the due date to give our Editor sufficient time to look over the document and ensure continuity among sections. It was his responsibility to request that a team member edit their own section to make it more continuous with the rest of the document, as well as make small edits to improve the overall writing quality.

3 Requirements

In this section we will survey the functional and performance requirements of our project. The functional requirements specify what operations our web app should be able to perform. This will be followed by the performance requirements (also known as the non-functional requirements), which are the intended quality attributes of PeakLearner.

We compiled this list of requirements in three distinct ways. The first was through our weekly meetings with Dr. Hocking, where we discussed the functional and performance requirements PeakLearner must meet. With this basis, we began to investigate more specific performance requirements using the developer tools for Mozilla Firefox and Google Chrome.

3.1 Functional Requirements

In this section we will consider the functional requirements for PeakLearner. These requirements are broken up into distinct categories to describe what a user can do with PeakLearner. Each use case is an operation PeakLearner can perform and covers one possible scenario a user may encounter. These functional requirements are structured in order of frequency, where the first requirement is the most frequently used and the last requirement is the least frequently used by an average user.

3.1.1 Interactive web interface

PeakLearner is a web app, and the basis of this webpage is a software tool called a genome browser. Biologists already use genome browsers to graphically show information about an uploaded genome. Since no current genome browser can both natively show multiple information files on a single "track" and allow for highlighting regions in the way that we need, we've augmented a browser to show the information that our servers generate.

3.1.2 Ability to add and modify labels

The ability for a user to add "labels" to the information is the basis of the machine learning that will generate the peak prediction models. These labels are created by biologists highlighting where peaks are and are not supposed to be. These are used by our backend server to judge how good a model is by marking if said model aligns with where a user says a peak should be. The one with the lowest error, the least amount out of line with the user labels, is the model we show. To do this, we need to implement two things. First, we need a way to show where a user has already added a label as well as differentiating what type of label has been added. Second, we need a highlighting tool for a user to be able to interactively add new labels as well as change existing ones.

3.1.3 Dynamic model generation

The ability to dynamically generate models is a critical feature to the system. peak models are simple graphical representations of where peaks should and should not appear along a genome. These models should be regenerated every time a new label is added to make sure the model displayed is the most accurate known model in the system. The dynamic model generation can be divided into two parts. First, we need to be able to pick the most accurate model out of the database based on the error matrix. Next, we need to be

able to determine a new lambda value to pass off to Monsoon. This model will need to be incorporated into our error matrix as soon as we get it back from Monsoon.

3.1.4 Upload capability

Scientists need to be able to upload their data to the PeakLearner system. This is one of the most important requirements because if a user can not add data, then they would have no reason to make models or add labels. Clearly, it is imperative that users are able to upload their own datasets into the system.

3.1.5 Download capability

The models that PeakLearner generates are useful only in comparison. The power of these models lies in comparing among many samples to identify which genes correlate to which diseases. For example, the model of a sample of someone with cancer is only useful when compared to one that is healthy. To facilitate this easily, we will allow a user to download a generated model as a bigWig and as a bigBed file for storage and future use.

3.2 Performance Requirements

3.2.1 Speed

Speed is the overarching requirement for this project, and with that in mind, we have come up with a few benchmarks that our web interface should meet in order to be deemed "fast enough." The expected overall time should be less than 0.2 seconds between new label input and showing the most accurate model to a user. If a user does not see immediate feedback on their input, they will be disinclined from adding more input to the system, which defeats the purpose of PeakLearner.

3.2.2 Simplicity of operations

An average user may add dozens of labels to a single dataset, making it one of the most important areas to optimize when designing out web application. We will use the following four approaches to simplify the process of adding a label and overall user interface.

3.2.3 Accuracy of model

Accurate model generation is another critical requirement of the system. Adding labels and using a simple interface is meaningless if a user does not get back an accurate model to fit all of the newly created labels. With that in mind, we have a plan to ensure we are always returning the most accurate model from our database. We are representing accuracy as a number of errors. Since the user is always right, we define an error as an instance when a given model contradicts a label set by the user.

3.2.4 Calculating new models

Sometimes there is no model with a perfect error score of 0 in the system. This is where NAU's cluster, Monsoon, comes into play. We will be using Monsoon to calculate new, and hopefully more accurate models for the database should we not be able to find an accurate model already in the system. However, Monsoon is a shared resource and we can not spend hours having it calculate millions of models, so we must be strategic with the data sent to Monsoon to not waste time calculating useless or duplicate models.

4 Architecture and Implementation

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. Finally, we will discuss how PeakLearner differs from what we intended to build.

4.1 Architecture

The key components of our system are:

- WEB BROWSER - 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 can properly see and interact with the data.

- WEB SERVER - 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 - stores past models and labels that were created on the web server. Models displayed on the web browser are stored in the database and passed to the browser through the web server.
- GPU CLUSTER - calculates 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.

4.2 Operational Flow

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

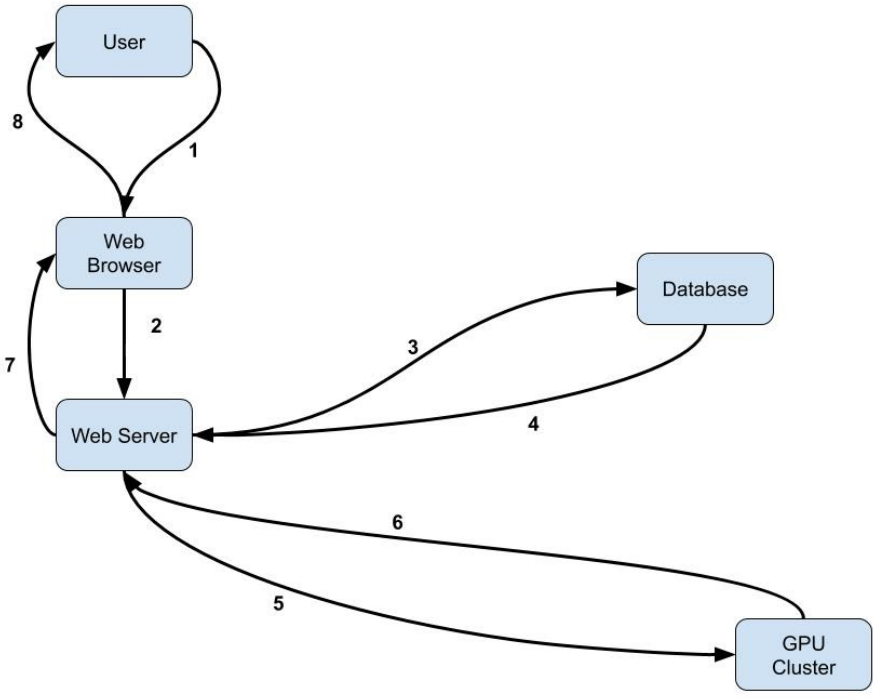

Figure 1: An overview of our system architecture.

1. Users interact with the WEB BROWSER by uploading their data. Users can 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 two.

2. Following 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 subfunction 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.

4.3 Module and Interface Descriptions

4.3.1 JBrowse

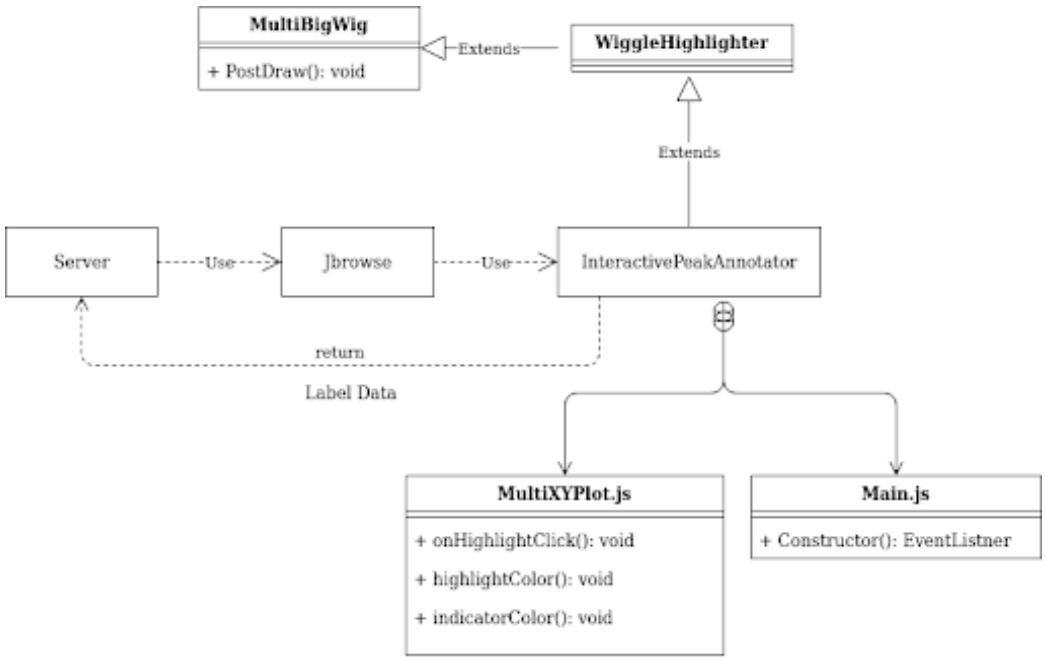

Figure 2: A diagram of JBrowse's functions.

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 was 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 sends 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 is in two other files that the back-end server needs to create. The first file is the model that we are 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 easy to interpret and much simpler to look at 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 the 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 have created the plugin InteractivePeakAnnotator (IPA) which extends the two plugins MultiBigWig and WiggleHighlighter. This plugin turns these two static plugins into a dynamic one. IPA uses the built-in highlight tool in order to add, manage, and remove labels.

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.

4.3.2 Server

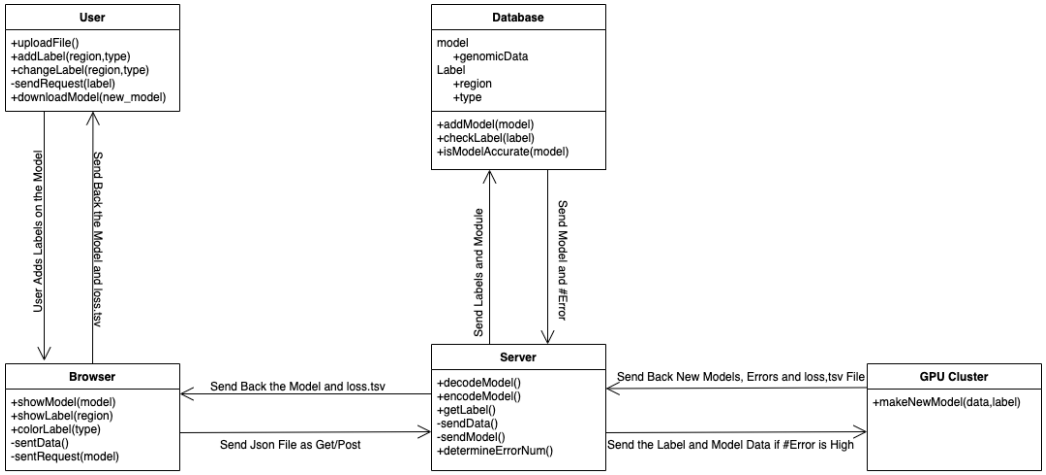

Figure 3: A diagram of the back-end.

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 and with the GPU cluster. 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 will then return a model and an error number This error number is essentially saying that the model that the database provided disagrees with exactly some number of the labels that were given. 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 as too high, then 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 and it is also 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 is taking in data from the browser, and directing it to the Database. Once the database sends back a model, then 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.This is clearly shown by the diagram above because of how the server is right in the center of the diagram and has all of the arrows pointing both to the server and away from the server.

4.3.3 Database

For this project we chose 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.

4.3.4 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 Testing

In order to adequately test our software, we will be implementing three different forms of testing to make sure every component is working as intended. Each of these tests will take place at a different level of the software.

5.1 Unit Testing

The first of the three levels will be the Unit Testing level. Unit testing is defined as testing at the lowest level, where a test covers a single unit of the code. These units can vary in size and complexity slightly, but a good unit test is usually about the size of a single function. These tests are done by developers while developing in order to make sure each function is working as intended.

Now that we have a base understanding of what Unit Testing is and why we are trying to use it, we need to know what tools are used to accomplish these goals. Since the majority of our server code is in Python, we will be using the Python unittest library to manage our unit tests. It was inspired by the common Java Unit Test library JUnit, and has a similar look and feel. Thus, unittest uses an object oriented approach to its unit testing, which is much easier to understand conceptually than a functional unit testing approach. The way unittest works is we import unittest and define tests on a per class basis. Then we run the test.main() function when we are ready to test our code and it will go through all of the predefined tests and run them against their predetermined answers. If any test of a function does not return the value that is expected, then that test fails and an error is displayed. Traditionally, all of this is done using the command line, but since we are implementing a continuous integration server, Travis-CI, these unit tests will run on every commit pushed back into our repository on GitHub.

5.2 Integration Testing

Integration testing is defined as testing the interaction of different units. For the scope of our project, integration testing will be done to ensure that the different pieces (browser, server, database, cluster) are working together as intended. For a picture of how these components are connected, the reader is referred to **Figure 1** in section 4.2.

To run integration tests and simulate communication between the server and the cluster, we used a Travis-CI environment that installed the SLURM scheduling software. Travis-CI uses a configurable .yml file that's located in our GitHub repo's topmost directory. It's in here that we specify the tests

to be run, the packages to be installed in the virtual environment, and the method used to notify us of the results. We decided to send a notification to a dedicated Slack channel in our workspace each time a commit was pushed to a specified git branch (either master or a testing branch.

5.3 Usability Testing

Usability Testing is the process to ensure that the standard user of a product will be able to easily access the features it provides. For PeakLearner our standard user is not someone with a programming background, but rather a biologist doing research on human genomic data. This means that all features we provide must be easily accessible through normal means, i.e. a mouse and keyboard, and not the command line.

There were two components of usability testing that we needed to test. First, we want to watch a user upload some data into the JBrowse plugin to make sure that it is intuitive and easy to use for scientists. Second, we want to be able to ensure that a user is able to easily add the correct type of labels on top of their data. To test these two cases, we planned to find three users, one who is another Capstone student, one who has some knowledge of biology and genetics, and a third user who is completely random. To assess the effectiveness of the testing, we will provide a URL to the user which will access our webpage. Next we will video call them using Zoom, and have the user share their screen with us. Finally, we will give the user a to-do list, like the one that follows below:

User TODO list

1. Open PeakLearner in a new tab
2. Upload this data
3. Add a Peak label
4. Add a NoPeak label
5. Delete your Peak label

We originally planned to have people sit down with a computer with PeakLearner installed and see how well they could use it. However, because of the lockdown we needed to adapt our testing strategy. We didn't want to have people need to install everything themselves as, if there were any issues, it would be hard for us to help, and it would be difficult to determine if the issue was usability or if PeakLearner was installed incorrectly. We decided to

share the screen of a computer with a known valid installation of PeakLearner with the test subject. Then we would go over what we wanted them to do verbally, then have the test subject give us instructions on what to do to complete the tasks.

Our test subjects had a range of familiarity with the subject. We tested several Biology and Computer Science majors for an assumed "familiar" group. We then tested several unrelated majors and people not in university for an assumed "unfamiliar" group. As we expected, the "familiar" group took less time to figure out how to use PeakLearner than the "unfamiliar" group. However, given the time to initially figure out how it worked the "unfamiliar" group was also able to start working competently with PeakLearner. The main issues seemed to be initially finding the highlight button and then how to change the type of label being added. However, it helped when a user was given a key of what the button looked like and what each label was colored. There's likely to be be a steep learning curve for picking up PeakLearner, but once a user knows how to use it they will be fine.

6 Project Timeline

There were six important phases to our project, as shown in **Figure 4**.

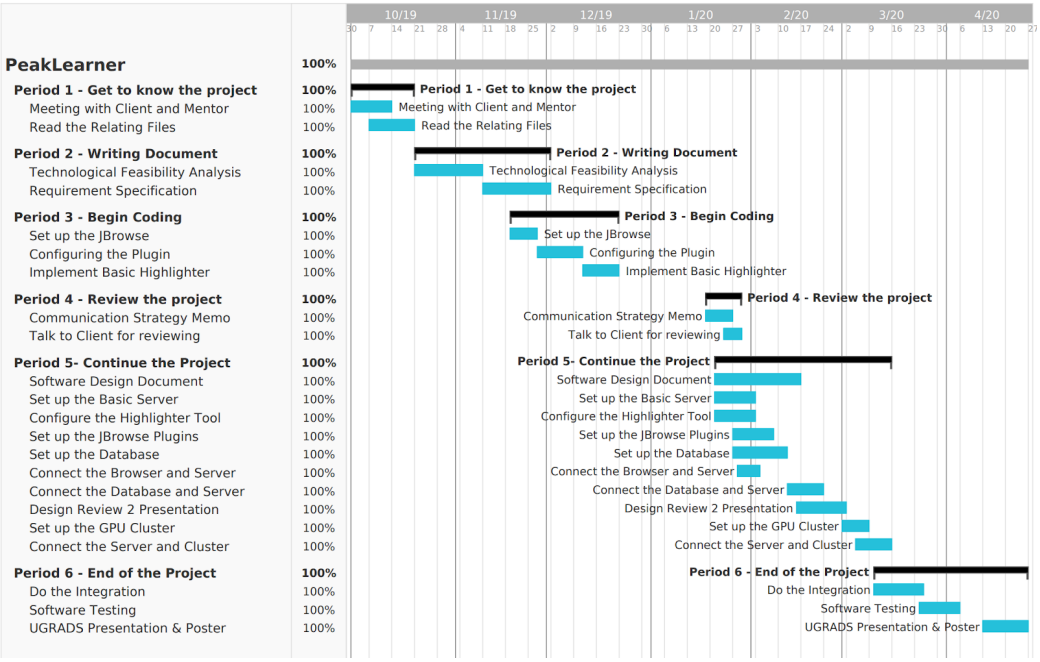

Figure 4: Our project timeline

1. *Understanding the project.* In this phase, we formed a team and held meetings with our mentor and client to get a preliminary understanding of our project. We also read relevant papers and literature to understand the knowledge related to the project.

2. *Preparation.* After getting a deeper understanding of our project, we started to consult the materials and write documents about the functional and technical aspects of the project.

3. *Beginning to code.* We had known about what we would do in this project and got a clear plan. So we set up JBrowse and implemented the basic highlighter tool on it. During this phase we consulted an outside expert on the team that maintains JBrowse.

4. *Reviewing the project.* After a long break, we spent two weeks reviewing and continuing the project. And we also made a detailed plan for the Spring semester. In this semester, we had more long-time meetings as a team and sub-teams which allowed us to really work together on the project.

5. *Continuing the project.* At this stage, we spent a long time on our project entity. We divided into two groups, one responsible for JBrowse and one responsible for the back-end. In the process of carrying out the project, we had weekly meetings with our client to ensure that the work direction was correct.

6. *Finishing up.* In the last stage of the semester, we conducted overall testing and unit testing on the completed projects. In addition, we also recorded the presentation to introduce our project and produced a project poster.

7 Future Work

Some of the future plans could include finishing the setup to allow the system to interact with Monsoon. This would be a significant improvement as the system could calculate new models much faster with access to a GPU cluster than it could as a single machine. A proof of concept test was done using Travis-CI to show the system is capable of running R code and getting back models, however due to time constraints we did not get permission to use Monsoon.

Another feature to implemented is the true/false positive matrix in the server. This is used to calculate the best model, and if one isn't found then send to the cluster to generate a new one. This should be done inside of the "send_Post()" method inside ourServer.py. This is the function that handles the incoming labels from the browser.

8 Conclusion

PeakLearner is 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 worked with Dr. Toby Hocking, a researcher at NAU who looks into how machine learning can support early detection of genetic diseases. He envisioned PeakLearner as a machine learning web app to help process genomic data for scientists. For Dr. Hocking, PeakLearner supports 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.

Glossary

bigBed : a binary indexed file format used in genome browsers for qualitative/categorical data. bigBed files are created initially from BED files, a plain text file format used in genome browsers for qualitative/categorical data. 7

bigWig : a binary indexed file format used in genome browsers for quantitative/real-valued data. 7

genome browser : a software tool for graphically displaying genomic data. 6

label : a graphical annotation marking regions of interest, including peaks. 6, 7

peak : an area of increased activity in a genome. 6

track : a sub-window in a genome browser displaying one sample. 6

9 Appendix A: Development Environment and Toolchain

9.1 Hardware

Our team developed the software primarily on a Linux Ubuntu 16.04 machine. However, the server can be run on any device as long as the necessary packages are installed properly. We have also tested the system on a Mac running version 10.15.4.

9.2 Toolchain

These are the languages and tools we used in our implementation.

9.2.1 Web interface

- languages: HTML, CSS, Javascript
- JS framework: Dojo
- JBrowse plugins: WiggleHighlighter, MultiBigWig, InteractivePeakAnnotator

9.2.2 Server

- language: Python
- libraries: NumPy, bsddb3

9.2.3 Database

- library: Berkeley-db

9.2.4 Cluster

- workload manager: SLURM
- R package: PeakSegDisk (`https://github.com/tdhock/PeakSegDisk`)

9.3 Setup

The setup process that we are about to walk through is for an Ubuntu 16.04 system. First install the following packages to prepare the system to run the backend server. Use apt-get install unless otherwise noted.

- nodejs
- berkeley-db version 6.1 (install this using brew)
- python3-bsddb3

Once all of these packages are installed properly, you will have everything needed to set up the server used to communicate with the web browser.

The next step is going to the PeakLearner GitHub repository, `https://github.com/yf6666/PeakLearner-1.1` and cloning the master branch, as it is the most stable branch. Once this is done, you will have to configure JBrowse to work properly, which is explained below. For additional information on setup and installation, read the README files in the jbrowse folder and the outermost directory.

9.4 Production Cycle

The source code can be found here to reinstall/setup the system: `https://github.com/yf6666/PeakLearner-1.1`

Separately, the majority of the action for the server is found in the ourServer.py file. The send_post is what handles adding labels to the database and is also where the code skeleton to hook up the system to a GPU cluster can be found. The majority of the server should be very stable, but any server issues can be handled in this file.

The code we have added to JBrowse can primarily be found inside of the main.js and MultiXYPlot.js files inside of InteractivePeakAnnotator. The links to the repositories that this code releases on can be found in the README inside of the JBrowse folder of PeakLearner.

Inside of main.js you will find the event listener to watch the highlight tool of jbrowse to create new labels. MultiXYPlot.js has the code to edit and remove labels.

The only other thing JBrowse will rely on would be the restAPI.py file. Note this is only used by tracks with the rest store class, including the setup discussed above for interactivePeakAnnotator tracks. For more info about the JBrowse REST API you can look here: `https://jbrowse.org/docs/data_formats.html`.