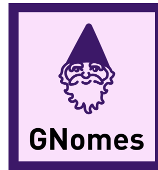

Software Test Plan

GNomes

Members
Jacob Christiansen
Allen Clarke
Yuanyuan Fu
John Jackson

Mentor
Mahsa Keshavarz

Clients
Dr. Toby Hocking
Christopher Coffey

Version 1.0
April 3, 2020

Contents

1 Introduction **2**

2 Unit Testing **3**

3 Integration Testing **5**

4 Usability Testing **7**

5 Conclusion **8**

1 Introduction

In the field of medical biology, a major topic of study is that of genetic diseases. This means researching diseases that are caused by our DNA, also called our genome. Genetic diseases are much different from normal bacterial or viral diseases. Instead of some invasion where we can simply try to eradicate the infection, a genetic disease is one that is caused by our own body, so it is more complicated to treat. We cannot simply eradicate the cause because the cause is our own body. Right now, we do not have a great understanding of what genes in our DNA code for genetic diseases. To solve this problem, we have been working with Dr. Toby Hocking to create PeakLearner, a machine learning web app to process genomic data. PeakLearner will take in genomic data and use labels set by biologists to determine where heavily activated genes are. These labels train an algorithm that predicts where other heavily used genes are as well. This will make a comparison between healthy and not healthy samples easier to understand and quicker to get to.

Testing for a project of this size is incredibly important, for multiple reasons. The first and most obvious reason is to make sure the software functions and runs in the environment that was planned. The other reason is to make sure the software actually succeeds in the features that were planned. This second reason is closely associated with the requirements that are determined at the beginning of the project. Overall testing plays an important role in making sure the product works and is the product designed at the beginning of the project.

In this document we will go over our plan for testing PeakLearner. To start we have created unit testing for our Python code. We have written our server and database in Python so it is imperative that they work as intended. Additionally, we have written some headless browser testing to test our front-end Javascript code. Finally, we have been using Travis CI to implement our continuous integration. This is to test that each time we push to our git repository that we haven't undone any other work and that the project overall is still working as intended.

2 Unit 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. 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.

The reason we are choosing to do unit testing is because it will give us the ability to confidently change our code later and know that the individual functions are still working as intended. This is a very important thing to know since if a function is changed and suddenly returns a value we did not expect, it could have significant consequences further along in the code. It also makes code maintenance easier in the long run since we can know exactly which changes broke which functions immediately, and not have to guess and hope we find which change caused a function to return the incorrect results. This is precisely the goal of unit testing. We want to be sure that every individual piece of the code works separately to ensure that they will all work when they are put together, and if a piece does not work, the problem is isolated to a single piece or function and is easy to fix.

2.1 Back-end

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.

We do not have many units to test on the server, as the majority of the server will be tested using integration testing (described below), but there are a few units that we need to test. The first unit we need to test is the decoder function that is used to decode the JSON we would receive from the web server and store it as a JSON object in Python. This is an important unit to test because we must be able to guarantee that we are able to decode our data and handle it correctly before storing it in the database. The other key unit test is to be able to correctly encode a JSON object that represents a model in preparation for returning the model to the browser.

Since the server is mostly just passing data along, its main jobs correctly encoding and decoding data as needed. There are, however, two edge cases for these respective functions. The first is the empty JSON object, and if the server were to decode an empty object, it should stop immediately and return to the browser, since it can't add an empty object to the database. The same goes for encoding models. If the server is trying to encode an empty object as a model, the browser will be confused about what the model is, so the server should stop immediately if it has an empty JSON object. The second edge case in both tests is an incredibly large dataset. We want to make sure we can handle large data as well as small data so that if the user sent 100 labels and we had to send back 100 models for 100 samples, everything still worked correctly. These are the main server unit tests and edge cases we plan on testing.

2.2 Front-end

Since the JBrowse UI involves click interactions, we will need unit tests that can simulate clicks. Though we've been testing PeakLearner GUI features as we go, i.e. by selecting a section of track and checking the console log, we need more structured tests that we can run as part of a testing suite to reduce time and cognitive load. We will be using the Mocha test framework for unit testing.

We want to make sure that when the user clicks and drags on a section of the browser track, information is correctly encoded in a JSON object. So, we need unit tests to simulate the action of creating a label. Two tests will check that the resultant JSON contains the correct values for starting position and end position. Another test will confirm that clicking on a label cycles through label types, changing the label type value in the JSON. One further test confirms that a fresh label has the same type as the last label. Finally, it's critical that when the browser receives a JSON object, it's correctly decoded and displayed to the track.

3 Integration Testing

Integration testing is the next of the three forms of testing we will be doing. This is one form of testing that we feel is most important because it tests how the individual pieces we tested above are going to be working together, and communication among components is the most important part of our project.

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. Referring to **Figure 1**, we will need an integration test for each of the 6 arrows below.

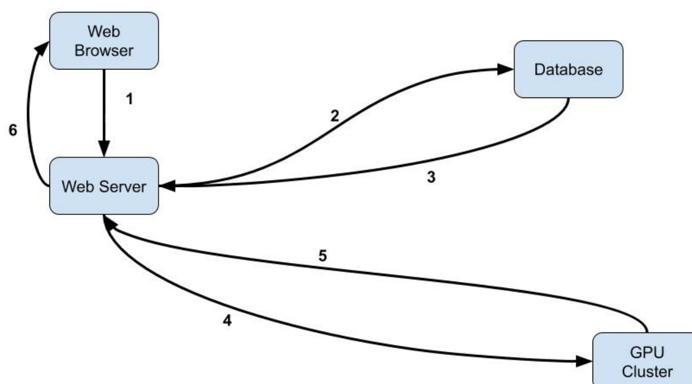

Figure 1: A diagram modeling the kinds of integration we will be testing.

These six integration tests will be critical to ensure that data is being transferred from place to place correctly and everything is working as intended. In order, the six tests are as follows:

1. Browser to Server. This test will test the browser encoding the JSON data of a label and sending it via AJAX to the server. The server will then decode the data and make sure it is exactly the same JSON object that was originally sent. We will know the test worked if the two objects match, which is easy to check since we are in control of both the browser and server using Travis CI. The input and output of this test are JSON objects that represent labels.

2. Server to Database. The test from the server from the database will test the server opening a connection to the database and inserting a label into the database. We will know the test worked by pairing it with the next test. The input for this test is a JSON label, and there is no output.

3. Database to Server. This test is partnered very closely with the previous test, and we will only be able to test if they both pass or fail in unison. This test will be one where the server opens a connection to the database and gets out a label. For testing purposes, we will get out the last label we put in and make sure that it matches the label we tried to put in the previous test. We will know that both these tests are working if the two labels match, as we are able to insert and remove data from the database correctly if that is the case. This test has no input and outputs a JSON label, and we are trying to match this output with the input of the previous test.

4. Server to Cluster. Our next text will test the communication between the server and the cluster. The purpose of this test is to see if we can send sample data to the cluster so it can be used to generate a new model. The inputs for this test are a set of labels and a set of data, both of which are going to be sent. This test is successful if the cluster is able to execute an R script with the data sent from the server. We will check this by comparing the inputs to the R script against some predetermined sample and label data to make sure they match.

5. Cluster to Server. The next test is partnered very closely with the previous test. It will feature the R script running with a predetermined set of inputs and the expected output should always be the same model (given the inputs). We will know the test worked correctly if the model returned from the cluster is sent back to the server and matches the predetermined correct model that we had found. The test is a success if the server receives back a model that is correct and matches the expected model. The test will fail if we get back an empty model.

6. Server to Browser. Our final integration test will test the communication between the server and browser. In this test the server will be encoding a model into a JSON object and returning it as a response to the browsers POST request. If the browser is able to decode and display the model in JBrowse, then the test passes, otherwise the test fails. The only input we cannot allow for this test is an empty model;

we know JBrowse can display this, but it is not helpful in any way to the user.

To simulate communication between the front-end and the server, we will need a headless browser. For this purpose, as part of our Travis CI suite we will use the Selenium framework, which is capable of launching a specified browser (such as Safari, Chrome, or Firefox) and conducting browser tests. Headless browser testing is critical for checking to see if PeakLearner is meeting performance benchmarks as well. As a reminder, we are aiming for a model-generation time of 0.2 seconds.

4 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. The standard user will be a biologist doing research on human genomic data. This means that all features we provide must be easily accessible through normal means. We are defining normal means by the methods one might use when web browsing—the mouse and keyboard. We cannot ask a user to rely on the console or URL bar to input commands.

We want to ensure that our users have an easy time learning how to use our product, and we are breaking the problem into two sections. First, we don't want to remove any usability that a user might expect. There are many tools that are used by biologists already and we don't want to just have a singular tool when normally a biologist might need many. Second, we want any novel features to be as simple to access and use as possible. We define simple as minimal mouse clicks in order to have the user's desired effect happen.

There are two components of usability testing that we need 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 plan 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

To ensure the testing is not biased, the GNomes team member supervising the call will not provide any feedback or instruction to the user unless they are stuck on one part of the list for over two minutes. We will know the system is effective if all three users are able to complete the entire list in under three minutes and without any guidance from a person with experience using PeakLearner. Since this is not a likely outcome, we will use the information we observe by watching these users to learn what pieces of the site are unintuitive and how we can improve them. Lastly, we can ask for user feedback after the users have tried to use the site to see what they felt was difficult or tedious.

5 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 software testing plan for PeakLearner, explaining to interested readers our goals for 1) unit testing 2) integration testing, and 3) usability testing. Overall we are confident that, with the testing plan we have compiled, we will ensure that our project fulfills its functions, yielding a product that allows biologists to have a greater understanding of our own genome.