LingoPros

Software Test Plan V.3

April 11, 2018

Josh Shaffer, Luis Montes, Erik Strauss, Matt Quintana

Sponsors: Dr. Okim Kang & Dr. David O. Johnson

Mentor: Ana Paula Chaves Steinmacher

NORTHERN ARIZONA UNIVERSITY

Overview: The software test plan document will explain the activities that will assure that our implementation exhibits the necessary functional and non-functional requirements. It will also ensure that our implementation is working as we expect it to and without error.

Table of Contents

Introduction 3

Unit Testing 4

Web Toolchain 4

AuToBI Toolchain 9

Integration Testing 11

ToBI Toolchain 11

AuToBI Testing 11

Neural Network 12

Web Toolchain 12

Usability Testing 15

Web Toolchain 15

Conclusion 17

Introduction

Dr. Okim Kang brought our team on to help streamline the research process that she does with her students at the Applied Linguistics Speech Lab (ALSL). They have an existing program in the lab that can perform speech analysis for them without the need for them to do it all by hand. However, the program is not able to be accessed by really anyone in the lab, so Dr. Kang wants us to put it online for her so anyone can access it as long as they have permission. Along with that, she also wants us to develop another speech analysis software using a specific framework called AuToBI. She wants this second program developed so they can compare their existing model with it and report their findings and validity of their software.

We have successfully developed both parts of the project and now all that remains is testing our programs. This is important so we know whether or not our project is working as intended and where it may be failing. As half of our project uses pre-existing code that we are not tasked to edit, there will not be as much testing on the web server as the outputs given are pretty much guaranteed to be what is expected. We really only need to test the security (such as user login and the special admin page we have given Okim) along with user specific information.

For the AuToBI portion of our project, however, there will be more more tests involved. We need to make sure the output from AuToBI is correct and we need to make sure that is put into our machine learning software correctly as well. The tests will check if the attributes that were created were really formed correctly and let us know if there were not. After that, we need to check if the machine learning is making accurate picks on those attributes to analyze and whether or not the final analysis is accurate. So unit testing will mostly focus on our machine learning software, whereas integration testing will focus on the transition from AuToBI to our own software.

Based on the previous outline, it's clear that one half of our project will have more tests than the other. This is simply because the web application relies on pre-existing code that we are not supposed to edit; we are essentially just putting it online for them. The AuToBI portion has more moving parts that we can actually go in and fix should there be errors. That's not to say we

won't be testing the web server, as there are things like database information that we need to verify. To make sure everything works as it should, we will implement various unit tests into our project which are outlined in the following section.

Unit Testing

We are only focusing on a subset of our system for unit testing because some of the methods or procedures are more appropriately tested in integration testing. The website portion of our project can be tested fairly well with unit tests as well as integration tests. The machine learning portion of our project using a Java API named Weka does not require unit testing because we are mainly using built-in methods that Weka provides for us so we assume they work. As for the AuToBI piece of the project we will be unit testing it as a black box and only worrying about the input and output of the program. The libraries we will be using to conduct our unit tests include JUnit for our Java program and the JavaScript assert module for our website.

Web Toolchain

The web application has particular functions both server side and client side that must be tested to work correctly and consistently or else the user's confidence in the system working as they need it to is in jeopardy.

Test Case 1: In order for accounts to be created they must have a valid access code as well as valid input fields.

Main flow:

1. User inputs their valid information into the required fields: First Name, Last Name, Username, Password, and Access Code.
2. User presses the submit button when they are done with input.

Expected outcome: The account is created and a message appears saying, "Success! Your account has been created."

Alternative flow:

1. User does not input valid information for one or more of the required fields: First Name, Last Name, Username, Password, and Access Code.
2. User clicks the submit button

Expected outcome: The account is not created and an error message alerts the user that something went wrong. Specifically it will say, “Error! Account not created.” and for any null field an error message appears saying, “Input a password” where the last word refers to the field that was null or invalid.

Test Case 2: A user must enter in valid and non-null account information to log in

Main flow:

1. User inputs a valid Username and Password.
2. User clicks the login button.

Expected outcome: The user successfully logs in and is navigated to the next page.

Alternative flow:

1. User does not input valid information for one or both of the required fields: Username or Password
2. User clicks the submit button

Expected outcome: The user is not logged in and an error message alerts the user that something went wrong. Specifically it will say, “Error! Incorrect Account Information” and for any null or invalid field an error message appears saying, “Input a valid password” where the last word refers to the field that was null or invalid.

Test Case 3: A logged in user must upload a .wav file in order to run analysis on the server. Null input or other file types are not allowed.

Main flow:

1. After logging in the user is on the home page.
2. User then navigates to the Upload File screen

3. User clicks “Upload File” and selects which file from his or her computer that he or she wants input to the server for analysis
4. User clicks the submit button to store the file on the server prior to analysis.

Expected outcome: A message appears saying the file has been uploaded successfully. The user is redirected to the Analyze File screen to begin analysis of that uploaded file.

Alternative flow:

1. User selects an unsupported file format to upload into the server.
2. User clicks submit.

Expected Outcome: The server rejects the file and displays an error message telling the user that the file format is not supported. For an empty submission, the website will display a message telling the user to upload a file before running an analysis.

Test Case 4: An input file for a logged in user must be analyzed on the server after the user clicks the arrow button in the Analyze File screen.

Main flow:

1. After the user uploads their file and clicks the submit button the file is stored on the server prior to analysis.
2. The user is then redirected to a page awaiting their confirmation to begin analysis.
3. After clicking the “Begin Analysis” button the file is then analyzed on the server and the user is redirected to the analysis results page.

Expected outcome: A message appears saying the file has been analyzed successfully and the user is redirected to the results screen that is appropriate for his or her account information. There on the results screen, the user will see his or her last submission and previous submissions organized by timestamp.

Alternative flow:

1. User clicks the Analyze File button but did not submit a file to upload for analysis prior

Expected outcome: A message appears reminding to them to go back to the “Upload File” screen, with a link to that screen, and to upload a .wav sound file before trying to start the analysis algorithm on the server.

Alternative flow:

1. User clicks the Analyze File button but the server timed-out in waiting for Praat to finish the analysis on the sound file.

Expected outcome: A message appears that tell the user the server could not finish analysis on his or her input file. The message suggests to try again with a link to that same "Analyze File" screen as well as suggesting to upload a smaller sound file with a link to take them back to the "Upload File" screen. The message makes clear to the user that exact problem is indeterminate as it could be an occasional breaking on the server's processing ability or that the file really was to big analyze.

Test Case 5: A logged in user must be able to access his or her past file analyses.

Main flow:

1. A logged in user can click a link to "See previous file analyses".
2. Upon clicking the above link, the use is taken to the "Previous Analyses" screen to see previous file analyses associated with his or her account.

Expected outcome: The "Previous Analyses" screen may or may not show file data and variables associated with the previously analyzed files depending on whether or not the user has had the server analyze any files yet. If he or she has analyzed with our service before, previous analyses will be shown organized by timestamps. There is no alternate flow because there is not any way for the user to do these steps wrong as they only click one link to see his or previous analyses and those analyses will either show if they are there or a blank table is presented to the user in the "Previous Analyses" screen.

Test Case 6: An admin account must be able to enter a valid access code to approve a new user's account creation.

Main flow:

1. The designated administrator goes to the login page and enters in admin credentials

2. Upon successful login, the admin then see his or her one screen: the “Access Codes” screen.
3. Admin then sees previously added access codes through a present table
4. Admin enters in the “Create new access code” field a new numeric code of any length but cannot be previously used.
5. Admin presses the field’s associated submit button after entering in a desired new access code and the access code is saved to our database for consumption by another user trying to make an account at some point in the future.

Expected outcome: The administrator is alerted by a message on his or her main screen that they successfully entered in a new access code to the database and upon page refresh will see that access code listed in the table of available access codes.

Alternative flow:

1. Admin inputs a null access code.

Expected outcome: A message will remind the admin that the access code cannot be null and asks the admin to try again with a link back to the “Access Codes” page.

Alternative flow:

1. Admin inputs an access code that contains a character that is not a digit.

Expected outcome: A message will remind the admin that the access code cannot contain non-numbers and asks the admin to try again with a link back to the “Access Codes” page.

Alternative flow:

1. Admin inputs an access code that is already in the table

Expected outcome: A message will remind the admin that the access code cannot be one previously created and asks the admin to try again with a link back to the “Access Codes” page.

AuToBI Toolchain

Test Case 7: In order for someone to run analysis through AuToBI they must run a command through the terminal and include an input .wav file and an output .arff file.

Main flow:

1. User enters in the command including a valid input and output file.
2. User submits the command

Expected outcome: The AuToBI analysis begins and the output file becomes populated with data.

Alternative flow:

1. User enters in the command without a valid input and output file
2. User submits the command.

Expected outcome: The analysis does not run and an error message displays in the terminal saying, “Error! The input file must be of type .wav” or “Error! The output file must be of type .arff”.

Test Case 8: In order for the AuToBI output file to be used in our Weka Java program it must be edited using our python script to correct issues with certain attributes. The .arff output will be passed into the python script for editing.

Main flow:

3. .arff results file is passed into the python script.
4. Attribute is found and type is changed

Expected outcome: An .arff file that has been fixed by replacing the original attribute type that was wrong with the correct type and is considered valid by Weka.

Alternative flow:

3. .arff file is passed into the python script
4. The attribute is not found.

Alternative flow 2:

1. .arff file is passed into the python script

2. The attribute is found but incorrectly changed.
3. Error is thrown to user stating that an error occurred in the processing of the .arff file

Expected outcome: The analysis does not run and an error message displays in the terminal saying, “Error! The .arff file has a data mismatch that was unable to be resolved.” or “Error! There is a missing attribute in the .arff data.”.

Test Case 9: The WekaRunner class contains a method, frequencyCounter() for determining which attributes are chosen most frequently over a series of analyses.

Main Flow:

1. The method takes in a series of files to run analysis on.
2. Each file is analyzed to determine which features were most important in characterizing it and an integer array representing the indices of the features is returned.
3. The analysis is repeated for all of the files and the count of how many times each feature was chosen is recorded.

Expected Outcome: After running all of the analyses, the frequencyCounter() function should return a list of integer values sorted by highest frequency to lowest from the analyses.

Alternative Flow 1:

1. The files are passed into the frequencyCounter() function.
2. An error occurs analyzing one of the files.

Expected Outcome: The user is notified of the error and with which file the problems occured.

Test Case 10: The neural network takes in an array of float values that correspond to the mean value of every attribute that was most frequently chosen in the Weka analysis.

Main Flow:

1. The array of float values is passed as the input layer of the neural network.
2. The input layer is multiplied by the weights on each input and passed to the second layer.
3. Values from the second are multiplied by their weights and passed to the output layer.

Expected Outcome: An array of four different float values that represent the probability of a certain proficiency level being estimated.

Alternative Flow 1:

1. While passing the data through the neural network, an error occurs in the matrix multiplication.

Expected Outcome: The passing of data is stopped and an error stated to user.

Alternative Flow 2:

1. No data in output layer

Expected Outcome: State to the user that there was an error with the neural net and that the analysis wasn't able to finish.

Integration Testing

ToBI Toolchain

For the ToBI toolchain that we have created, there are three modules that pass data from one to the other: The AuToBI program, the Weka feature selection and the neural net. These are the modules that connect together to form the full toolchain.

AuToBI Testing

Integration Test 1: Check the output of AuToBI for valid file data
Input: Pass in .wav file to AuToBI program
Expected Outcome: The AuToBI program should produce an .arff file which contains an attribute and a data section of the file. The attribute section should have a series of attribute names, and the data section should have a series of data arrays, where each entry corresponds to a measurement of the respective attribute.
Alternative Outcome 1: The file produced has no data inside of it. State an error to the user that there was an issue with the data generation and stop the toolchain.
Alternative Outcome 2: The AuToBI program does not finish running at all, and no file is produced. State an error to the user that the program failed to completely run and stop the toolchain.

Weka Testing

Integration Test 2: Check that Weka analysis determined a set of important features

Input: Pass in the .arff file to Weka analysis portion

Expected Outcome: The Weka analysis should return an array of integers that represent the set of indices that it deemed most important to characterizing the data.
Alternative Outcome: No integer array is returned at all. State to the user that there was an error processing the file. Stop the toolchain.

Integration Test 3: Check that Weka analysis produces the mean value of each attribute.
Input: Integer array of attribute indices
Expected Outcome: An array of float values that represent the mean value of each attribute.
Alternative Outcome: No mean array is returned at all. State to the user that there was an error calculating attribute values and stop the toolchain.

Neural Network

Integration Test 4: Check that the neural network produces a guess about the proficiency
Input: Array of float values that represent the mean values of each chosen attribute
Expected Outcome: The neural net must output a guess of the proficiency level in the form of an integer.
Alternative Outcome: There is an issue with the calculations and no guess is made. State that there is an error with the network and halt the toolchain.

Integration Test 5: Check that the accuracy of the net's guess is above a certain threshold.
Input: Series of training and test files to the program.
Expected Outcome: A trained neural network that makes correct guesses on more than 70% of test cases.
Alternative Outcome: The network fails to make correct guesses in 70% of cases. Re-do training and modify the network calculations.

Web Toolchain

There are four major components that must work together consistently and correctly for our minimum viable product. These major components are the Node.js server, the client side interface for the Node.js server, the MongoDB database, and the Praat script. There are many

possible entity relationships that could fail or be unsafe here, but we can come down to around five possible major issues that we can test for.

Integration Test 5: The Node.js server could stop sending the client data: We can test to make sure the client receives data from the server every time someone's browser is pointed to the DigitalOcean IP space by writing in a server logged message each time a client requests something from it and the client got a return of anything. That way, the data of anything that gets returned to client will prove that the server is talking to that client.
Input: A client socket connection to the server.
Expected Outcome: A server logged message that a user has connected.
Alternative Outcome: No server logged message, then signal to the user that server is not listening to them.

Integration Test 6: The MongoDB database may stop communicating with the server: We can test to make sure that server and database are talking to each other by asserting a specific selected account's first name lookup originating from the server, and expecting an exact return message from the database based on what we can expect from that selected account's firstname being looked up which is its firstname and its Mongo return signature.
Input: A string of lookup query for a specific account's firstname for our database sent to the database we can trust to be in the database because we put it there.
Expected Outcome: A specific database response string.
Alternative Outcome: The expected response string is not the same as the actual response string and that would indicate to the site maintainer that the database and server are not communicating and he or she needs to restart "mongo" the program and try to pass this test again.

Integration Test 7: The server may fail to generate a Praat instance needed for a file's analysis using Node.js: We can test to make sure the Praat instance can happen by asserting a child process of Praat on the server and expecting the return stream string from the Praat environment

starting up in that test. That way, we can know that the server can start Praat and that file entry to Praat producing incorrect results means a problem exists with the Praat script rather than the server.

Input: Command to start a child instance of Praat.

Expected Outcome: Stream from the Praat instance confirming that it is running on the server.

Alternative Outcome: Error in the creation of the Praat instance which will return a string notifying us of its failure.

Integration Test 8: The interface may fail to send correct types of input data to the server: We can test to make sure that the client sends correct types of input data by asserting per user input field that a particular type came back even after irregularly asserted user input data. For example, the access code must be entered as an integer and anything else must be turned into an integer. We would test that the access code field has input casted to an integer by asserting access code entry with strings, null (empty), or booleans and expecting a returned type check of integer. If the return fails to be of type integer, we know we didn't cast that field's entry to an integer. We could do these kind of assertions for every user input field.

Input: Input information in each field and have it sent to the server.

Expected Outcome: Information is all correctly casted as their respective types and the server accepts it.

Alternative Outcome: At least one field has an incorrect type in it and the server rejects it. A status message will confirm which field is incorrect.

Integration Test 9: The interface may send dangerous data to server (script tags, database drop statements, links etc.): We could test that our user input fields have properly been sanitized by making assert statements per field with such dangerous data as suggested and expecting return statements of stripped out script tags, stringify drop statements, and stripped out link tags. Upon seeing sanitized return statements from the user fields to the server side, we can predict lower likelihood of our web application being able to be maliciously penetrated. If we see unsanitized

return statements (i.e. the tags scripts and links are still present), we know our sanitizing mechanisms are not working.

Input: Every field has dangerous input submitted through them like strings containing script tags or strings with system commands like “exec” (for server side execution) followed by Mongo table manipulations (like “.drop” or “.insert”).

Expected Outcome: Those same strings sent to the assert statement but with script tags and system calls removed.

Alternative Outcome: If the strings remain the same or remain potentially runnable on our server than we know our inputs are not as safe as they should be and we need to try a different input sanitizer library or make our own.

Usability Testing

Web Toolchain

Users have five major possible actions they must be able to navigate to and use on our web application very intuitively. Those five actions are to create an account, log in, upload a sound file for analyzing, see previous analyzed results, and if they are the admin: be able to make access codes. On top of those five actions, our company has sought to make aesthetically pleasing yet simple pages that users would be interacting with and to that end we need to test that the user liked what they saw and thought it was

The expectations as previously explained can be tricky to test. Unlike a right or wrong outputting function, there is a range of acceptable user experience for the web application. To test whether we’ve made an intuitive and stylish product we need to leverage a focus group. We can do this by asking Dr. Kang and at least three of her Linguistics students to try to conduct the first four actions and Dr. Kang to conduct the fifth the access code generation action. After about 10 minutes of feature attempting, we can have them fill out a survey.

The survey would ask users three questions. The first question to users would be to rate on a scale of 1-10 (10 being most intuitive and 1 being least) their ability to understand how to use each of the actions pages (login page, create account page, upload file page, etc.). The second question to users would be rate on a scale of 1-10 (10 being most stylish to 1 being least) how great the look of the web application was to them. The third question to users would be to rate on a scale of 1-10 (10 being most logical and 1 being least) how logical the linking of the web application pages was.

We would then look at the average scores per question of each of the three. Then iff our threshold minimum average was greater than the actual average of the question, we would know that in that questions category we should try changing that particular aspect of our web application until a new round of surveying would indicate we met or exceeded our expected question average.

Four total scores on the first question averaging at least seven would be passing for us to know that our action pages were straightforward to use. Four total scores on the second question averaging at least 5 would be passing to know that our aesthetics were decent enough to present at UGRADs in May. Question two's threshold can be that low because none of our team are digital artists and excellent aesthetics are not required for meeting client requirements per our Requirements Document. Four total scored on the third question averaging at least 8 would be passing to know that our navigation made sense and was traversable.

Conclusion

We now have a roadmap to test our project in order to ensure reliable and usable software for our client. Our software can be counted on to both work as stated as well as allow users to understand their mistakes and point them in the right direction when an error occurs while using the product's features. While we still have to write the code to accomplish these tests, we know in a detailed perspective what needs test coverage and what the goals of these tests are.