AGTC

genetic tax. consultants

Final Software Testing Plan

Christian Buskirk
Peter Bellagh
Chris Blazer
Jorden Kreps
Curtis Rose
Faculty Mentor/Project Sponsor Viacheslav “Slava” Fofanov, PhD

Table of Contents

1. Glossary

Big Data - extremely large data sets that may be analyzed computationally to reveal patterns, trends, and associations, especially relating to human behavior and interactions.

Black Box Testing - a software testing method in which the internal structure/ design/ implementation of the item being tested is not known to the tester.

Configuration File - A file that contains data about a specific user, program, computer, or other file. They are generally read at startup by the operating system and other applications in order to customize the environment for the user.

HPC Cluster - A “High-Performance Computing Cluster”; a set of connected computers that work together such that they can viewed as a single high performance system or “supercomputer”.

Luigi - A Python module that helps build complex pipelines for batch jobs. It handles dependency resolution, workflow management, and visualizations. This is the pipeline management tool that we have selected for this project.

Orchard - A pipeline creation tool with a command-line interface. Assists in creating proper branches and file structures across multiple runs of the same dataset.

Pipeline - A pipeline consists of a chain of processing elements arranged so that the output of each element is the input of the next.

State System - A system/program that can be in one of a finite number of states. The system/program can transition from one state to another. In our case, the system can transition, linearly, from one state to another while saving the output of those states into files.

UML - Unified Modeling Language, a general-purpose modeling language used to provide a standard way of visualizing the design of a system.

White Box Testing - a method of testing software that tests internal structures or workings of an application, as opposed to its functionality

2. Introduction

The AGTC Genetic Tax Consultants set out to create a tool that facilitates multiple runs of a big data pipeline; allowing each run to use different parameters and still retain all variants of the inputs and outputs in a neat and orderly fashion. This Big Data tool is named Orchard and is a command line application that deals heavily with the user's file system.

Orchard is a highly modular tool that wraps around the pipeline manager called Luigi and in order to make sure all components are working together correctly, a large multitude of tests need to be implemented. Each component of Orchard is directly dependent on the outputs of one or more other components, and they need to be in synch for them all to work. Orchard has 4 main components, the Driver, the Configuration File Reader, the Branch Manager, and the Luigi File Generator. Together, these components will take file paths to configuration files from the command line, which are entered by the user, interpret them, and produce Luigi files that represent the pipeline. Orchard will eliminate the need for the user to directly interact with complex luigi files, and allow them to simply input the variables they want into an easy to use configuration file. Orchard will create these Luigi files and run the pipeline.

In order to test the efficacy and ability of Orchard, we will need to set up large variety of tests that insure everything is working as we expect. To ensure that every aspect of Orchard is tested, we have a 3 part testing plan in place to test our product thoroughly. Our first phase of testing will be white box Unit testing. These tests will be inserted directly into our code to insure that we have 100 percent code coverage and to eliminate any bugs and errors that would come as a direct consequence from our coding. Orchard will be a test driven project using Python's built in unit testing package.

Next, we will begin Integration Testing, which is meant to test the interactions between the Orchard components themselves. We need to be sure that all of the individual components are interacting with each other as expected because each components is directly dependent on one or more of our other components. These components need to be outputting the correct data and file types to the other components for everything to run. All of these tests will be black box tests, meaning they will not be tests written into our program, but tests done by using our program and observing results. We will be trying to break our program in any way we can think, and then use those results to improve Orchard's durability.

Finally, we will begin usability testing where we will observe test users applying our software to actual problems. For these tests, we will observe a variety of users with a range of technical abilities. One of our main requirements was to deliver a product that

could be used by people with limited computer science background. It is important that when we, as a team, graduate, and leave Northern Arizona University, that our software can still be used and modified. Luigi is a complex and difficult tool to use, and Orchards main goal is to make it easier to use the pipeline manager. Through our tests and observation, we hope to increase Orchards ease of use and the information we gather will directly influence the design of Orchard through any and all needed modifications.

3. Unit Testing

We are aiming to cover all of our main components with unit tests to provide us a robust test driven environment to work in. Having these main components covered will ensure that any errant code does not get pushed into our repository without having been tested and confirmed to be an error.

We also aim to have one-hundred percent code coverage of the entire project before we are finished. This is not so much for ourselves as it is for the clients of the software. They may need to add more code into our software and we want to ensure that there is good coverage for them. This, as well as code documentation, will help the owners update the code when and if they need to.

3.1. Testing Plan

Since Orchard is written in Python, it comes with it's own unit-testing suite, simply named unittest. This allows either class-based or functional testing for any and all Python code in the package. Each package and subpackage within the Orchard source code will have a directory called tests, and in that directory there will be a test_(name).py, where each Python file in the package, or subpackage, will have their own testing file to keep everything organized.

The component used to track code coverage is also simply named coverage, and is available through PyPi. This relies on a .coveragerc file telling it which files to ignore, and test for coverage. The driver that will run the tests with coverage is a PyPi package called pytest (or nose, as either package would work the same).

The combination of all these packages will allow our continuous integration testing through GitHub hooks to TravisCI, to publish and track our coverage percentage through another free service called Coveralls. This adds two badges to our repository, one saying if our tests passed, and another saying how much of the code base is covered by tests.

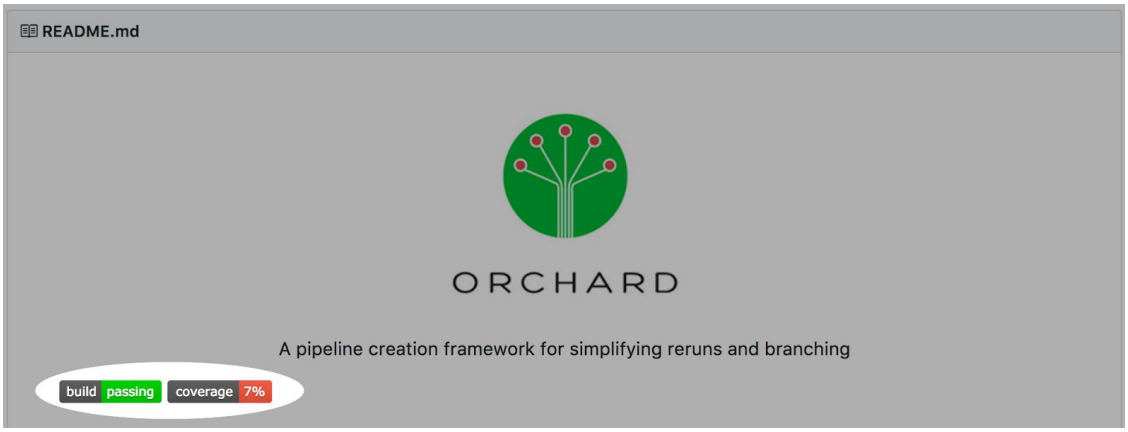

Figure 1 - Orchard README Showcasing Travis and Coveralls Badges

As stated earlier, each package will have it's own dedicated test suite. Orchard itself has been split into two major pieces, the pipeline driver, and the configuration template creator, which will both be covered in detail below.

3.1.1. Driver

The Driver component is what takes the pipeline from configuration file, to a runnable Luigi file written in Python. It consists of three main parts, the configuration file reader and validator, the branching manager, and the Luigi file creation itself.

- Configuration File Reader:
  The configuration file reader has two parts at the moment, the validation tool, and the *simplify* method. Validation will be tested by passing it a link and configuration file, and making sure they pass validation. This will be cross-tested with files that are known to fail to make sure the validation catches the errors. *Simplify* takes a filled-out configuration file and strips it down to what should have been generated by the link file, and this will be tested by passing it a filled out configuration file, then testing it against a known correct or incorrect file.

- Required Variable Input:
  Variables that are required must be entered. There will be two tests, one where all the required variables are entered and it results in a pass. The other test will be missing several required variables and if it errors, results in a pass.

- Configuration File Format:
  When a user generates a configuration file template from a link file, they use that file to fill out their variables necessary for the pipeline to operate.

There is a validation done at this point to make sure the user didn't mistakenly remove critical portions of the configuration file. This section will be tested several times with the valid configuration file. It will also be tested several times with portions missing to make sure it is creating errors appropriately.

- Required Exclusive Variable Input:
  Exclusive variables in a group where one, and only one variable can be entered. There will be three tests. One test will produce an an error if no arguments are entered and result in a pass. The next test will pass if one and only one argument is given. The third test will produce an error if more than one argument is entered and result in a pass.

- Optional Variable Input:
  Optional variables are variables that are not required for the Orchard components to run, but may be entered into the configuration files. Optionals are not capable of being tested. They can be entered or they can not be entered. They will be tested in integration testing to make sure they operate as intended.

- Optional Exclusive Variable Input:
  Optional exclusive variables are those that don't need to be entered, but if they are, one and only one can be entered. With optional exclusive input there will be three tests. One test will pass if no optional exclusive variables are entered. The next test will pass if only one of them are entered. The last will result in an error if more than one are entered and will result in a pass.

3.1.2. Branch Handler

The branching handler will be tested by giving it a fake workspace in a temporary directory, and passing it configuration files to see if it correctly branches, keeps its track, or knows that the run has already been completed.

- Directory Creation:
  Directory creation will be tested 3 times. The first test is to see if, on a clean version of Orchard, a directory gets created in the Orchard directory when the software has never been run before. The next test will ensure that a branch occurs if there is a similarity between the first pipeline and the current pipeline and a directory should be created within the first. The last test will check if a new pipeline that has no similarities to the first two creates a sub from the Orchard directory parallel with the first test.

- File Paths:
  We will test 3 times that file paths are in the resulting configuration files are correct. The three cases will be similar to the three above but will specifically be checking the resulting configuration files and their updated file paths.

- Symbolic Linking:
  To make sure that the users have all of the outputs of a run in one location we test (but not wholly copied to save time and storage), in similar circumstances to the three tests described above, that symbolic links are created when applicable and are not created when they are not needed.

- Luigi File Generator:
  The Luigi file generator takes the pre-validated configuration file, and turns it into valid Python code that will be ran by the Luigi pipeline manager. This will be tested by feeding it configuration data and testing it against expected outputs, and also by attempting to run the luigi file itself to make sure everything is valid in that respect.

- Luigi Files:
  We will test 4 instances of unique pipelines to test that luigi files are being generated appropriately. These four tests will use combinations of required, optional, exclusive, required-exclusive, and optional-exclusive variables. We foresee four tests being all that is necessary although more may be needed for combinations of those four variables types.

3.1.3. Configuration File Template Generator

The configuration file template generator takes a link file and outputs a simplified version of it to be filled out by the user. This is a simple component, but it important in its own right, as an incorrect configuration file will cause the program to work incorrectly.

- Template Creation:
  This component takes a link file, and strips it down to a configuration file. This will be tested by passing it valid link files and comparing it with expected output configuration files, as well as passing it invalid link files, and making sure it fails properly, as leaving keys that aren't valid in the configuration file will cause failures further down the line.

- Configuration File Template:
  There will be two tests to determine if the configuration file template is being generated appropriately. One test will be for a link file containing very few modules without special cases. This is a simple test that should only fail if there is something critically wrong with the code. The second test will be a larger link file, containing three or more modules that all contain special case parameters, such as flags, optionals, and exclusives, which should comprehensively test this component.

4. Integration Testing

This section details the black box tests needed to ensure that each component of the Orchard Pipeline Manager are interacting with each other as we are expecting. Black box tests are executed through the use of our user interface and tests built directly into our code. These tests are extremely important in the context of our project, because as a pipeline manager, the inputs and parameters of each component are directly dependent on the outputs from dependent components.

4.1. Configuration File Reader and Luigi File Generator

4.1.1. Purpose

The purpose of this test to ensure that once the configuration file is generated from the Configuration File Reader, that the configuration file is passed to the Luigi File Generator, and is in a format which the Luigi file generator can read, so that it can be turned into a luigi file without the user being involved.

4.1.2. Test Description

- Have a user enter a file path to a file that does not exist to see if the system returns the correct.
- Have a user fill out a configuration file with the incorrect file type and track the errors
- Have a user fill out a configuration file of the correct file type and enter the path into the Configuration File Reader
- Track output file type, see if it matches the file type needed for the Luigi File Generator.
- Track Luigi File Generator output, see of input was accepted, processed, and a corresponding luigi file was outputted.

4.1.3. Expected Results

We expect the output of the Configuration File Template Generator to be a .yaml file, which is what the Luigi File Generator is expecting as a parameter. We then expect a luigi file with the corresponding values entered into the configuration file to be outputted from the Luigi File Generator, which will be a python file (.py).

4.1.4. Conditions Required To Pass Testing

To pass it's integration testing, The Configuration File Reader and The Luigi File Generator must pass and accept the file paths of the correct file types, and throw errors when the file type is that of a different file type, or does not exist.

- A path to a .yaml configuration file must be passed from the Configuration File Reader to the Luigi File Generator
- If the Configuration File Reader passes a file path to the Luigi File Generator that is not of .yaml file type, an error must be thrown
- If the Configuration File Reader passes a file path to the Luigi File Generator that does not exist, an error must be thrown

To be considered successful, all three of these tests must be passed.

4.2. Configuration File Reader and Branch Manager

4.2.1. Purpose

The purpose of this test is to ensure the interactions between the Configuration File Reader and the Branch Manager works as expected. The Branch Manager's input parameters are directly dependent on the outputs from the the Configuration File Reader's output. The output from the Configuration File Reader is a file with the data taken from the inputted configuration file, which the branch manager reads, and determines if a branch needs to be made depending on the variables it receives as well as stores the branches in the correct file system

4.2.2. Test Description

- A configuration file of the right file type is entered into the Configuration File Reader, and the output is tracked to ensure that the data contained within the configuration file is readable by the Branch Manager.
- Once the first test has passed, the user will enter configuration files testing each of the individual variables and flags possible in the configuration file to see if the branch manager makes the correct branching decisions.
- Then the user must check to make sure that all of the branches were stored in the correct files for organization and reuse.

4.2.3. Expected Results

We expect that if configuration files with missing or incorrect parameters are entered, that the branch manager catches this and returns and error to the user. Once the correct files are being entered, the branch manager will correctly interpret the variables and flags in the configuration file and make the appropriate branches when needed. These branched runs should then be stored and named correctly in the appropriate file structure for easy tracking and readability.

4.2.4. Conditions Required To Pass Testing

To pass it's integration testing, The Configuration File Reader and The Branch Manager need to have certain interactions when it comes to file passing and error handling. For these components to pass their integration testings:

- A path to a .yaml configuration file must be passed from the Configuration File Reader to the Branch Manager
- If the Configuration File Reader passes a file path to the Branch Manager that is not of .yaml file type, an error must be thrown
- If the Configuration File Reader passes a file path to the Branch Manager that does not exist, an error must be thrown

To be considered successful, all three of these tests must be passed.

4.3. Driver and Configuration File Reader

4.3.1. Purpose

The purpose of this test is to determine if the Driver correctly sends the configure file path and link file path from the command line to the Configuration File reader in a format that the file reader can interpret. The user will enter these two file paths in the command line and the file reader component will then be able to correctly get the data from these files and output the data to the configuration file template generator.

4.3.2. Test Description

- A user will input many different incorrect file types to test the error detection capabilities of the Configuration File Reader
- A user will then enter file paths pointing to files that do not exist in an attempt to trigger the correct error response from the Configuration File Reader
- The user will then enter in the correct file types, and track whether the Configuration File Reader accepts the files, and can read the data off of said files
- The output of the Configuration File reader will then be tracked to ensure that the files were read correctly, and the output data was in the correct form to be passed onto the Luigi File generator

4.3.3. Expected Results

The Configuration File Reader should return an error to the user if the user inputted file paths were paths to files of the wrong type, displaying to the user what the expected file type was, and what the actual file type was. Also, if the file path pointed to a file that didn't exist, another error would be triggered alerting the user that the file does not exist. If a correct file path was entered, the Configuration File Reader should be able to read the data, and pass it onto the Luigi File Generator in the .yaml format

4.3.4. Conditions Required To Pass Testing

To pass it's integration testing, The Driver's input must be sent to the Configuration File Reader, and The Configuration File Reader must check that the command line arguments passed are file paths to the correct file type. . For these components to pass their integration testings:

- The user enters a file path on the command line that is path to a file that is not a .yaml file, and an error will be returned to the user informing them that only files of .yaml type are accepted
- The user enters a file path of a file that does not exist and an error must the returned to the user informing them that the command line argument must be a file path to a .yaml file
- The user will not enter anything into the command line as a parameter, and an error must be returned informing the user of what the expected parameters are.
- A user will enter a file path in the command line that leads to a .yaml file, and the configuration file reader uses this parameter to read the configuration file that was entered and record the needed to data.

To be considered successful, all four of these tests must be passed.

5. Usability Testing

Our usability testing plan consists of two parts, testing the roll-out of Orchard and testing it's ease of use among users of varying technical skill levels.

5.1. Ease of Rollout

Rollout will be done using pip. Testing will be handled mostly internally, with some external user testing. We will test our rollout by installing our Orchard package through pip on the following major releases of the following operating systems:

- CentOS
    - 7.1
    - 7.2
    - 7.3
- Ubuntu
    - 16.10
    - 16.04
    - 14.04.5
    - 12.04.5

We will have 3 users with different levels of unix fluency test rolling out Orchard. Proper documentation should allow anybody to install Orchard through pip. In order for this test to pass all 3 users must be able to install Orchard by following our documentation.

5.2. User Testing

5.2.1. Test Composition

User testing will comprise of four parts: internal testing and high, medium, and low technical skill level user testing. Users in each category will test the following 4 parts of Orchard:

- Adding and removing components
- Submitting a configuration file
- Starting the pipeline
- Branching the pipeline

The users will be giving our documentation and we will test if they can complete the aforementioned tasks using only our documentation. All issues found during usability testing will be resolved by simplifying our interface and expanding our documentation and tutorials.

5.2.2. Internal Testing

We will do extensive internal testing. This will help us write documentation designed for an end user to be able to use Orchard. We will follow our documentation in testing implementation to find out if it is sufficient.

5.2.3. High Technical Skill Level User Testing

We will have 2 users with a high level of technical skill test Orchard to make sure they can implement it based off our documentation. Users from this section will consist primarily of our mentor Professor Fofanov himself. He has the most in-depth knowledge of the existing pipeline and highest level of skill to be able to implement Orchard. In order for this test to pass both users must be able to implement Orchard by following our documentation.

5.2.4. Medium Technical Skill Level User Testing

We will have 2 users with a medium level of technical skill test Orchard to make sure they can implement it based off our documentation. Users from this section will consist of Professor Fofanov's associates; students who work in his lab. They have knowledge of the existing pipeline and some programming experience. This is the most important user testing step because this is our expectation of the typical user for Orchard. In order for this test to pass both users must be able to implement Orchard by following our documentation.

5.2.5. Low Technical Skill Level User Testing

We will have 2 users with a low level of technical skill test Orchard to make sure they can implement it based off our documentation. Users from this section will consist of associates of ours from within the Computer Science program and Professor Fofanov's associates; students who work in his lab. Some of the lab aids have limited programming experience, and our associates have no knowledge of the existing pipeline. This step exists to ensure the clarity of our documentation and ease of use of Orchard. In order for this test to pass at least one user must be able to implement Orchard by following our documentation.