Software Testing Plan

Version 1.0

Jet Propulsion Laboratory Image Analysis

Client: Iona Brockie

Mentor: Austin Sanders

Team Hindsight

Charles Beck, Alexanderia Nelson, Adam Paquette, Hunter Rainen

March 31, 2018

1. Introduction 2
1.1. Purpose 2
1.2. Project Overview 2

2. Testing Framework 3

3. Unit Testing 4
3.1 Image analysis (Model) 4
3.2 Graphical User Interface (View) 5
3.3 Configuration 5
3.4 Control (Controller) 5

3. Integration Testing 7
3.1 View to Controller 7
3.2 Controller to Model and Model to Controller 8
3.3 Controller to View 8

4. Usability Testing 10

1. Introduction

1.1. Purpose

This test plan describes our testing approach and framework for testing our automatic image analysis tool for NASA JPL. This document introduces:

- Unit testing: Testing small independent pieces of our tool to ensure reliability and accurate functionality.
- Integration testing: Testing the interfaces between major modules and components to make sure interactions and data exchanges occur correctly.
- Usability testing: Testing to to ensure our end user (JPL) can understand and use our tool effectively to produce quality analysis.

1.2. Project Overview

We are Team Hindsight, and we are working on the project 'Image Analysis of Abraded Rocks to Determine Dust-Free Area'. Our project sponsor is the Jet Propulsion Laboratory (JPL) at the California Institute of Technology with our main point of contact/client Iona Brockie, a Mechatronics Engineer at JPL.

JPL is a federally funded NASA research and development center whose primary role is to construct and operate planetary robotic spacecraft. Mars is one planetary body that JPL has been actively carrying out scientific missions on over that last few decades. JPL's newest rover, Mars 2020 (M2020) will aim to further deepen our understanding of whether or not the red planet was once habitable. Specifically, the primary goal of M2020 will be to look for evidence of past life on Mars by analyzing and collecting samples of the Martian surface by drilling into rocks, which will then be picked up by a future mission.

In its quest to find evidence of past life, the M2020 rover will use a suite of tools including an onboard drill with a set of drill bits to take measurements of the soil/rock and potentially collect samples from the Martian surface to return to Earth later. To identify what samples to take, the rover is equipped with a Planetary Instrument for X-ray Lithochemistry (PIXL) camera. The PIXL camera looks at a particular region and analyzes it for chemical compounds and elemental makeup. However, before instruments like PIXL can analyze samples, the rover needs to overcome a problem inherent to drilling. When the rover drills into a rock, it creates a lot of dust, obscuring the hole. JPL then blows dust out of the hole using compressed gas.

Team hindsight's goal is to create reliable software to automatically analyze the effectiveness of this gas Dust Removal Tool (gDRT). Our tool will look at before and after images of dust being blown out of the hole and return the area covered in dust after using the gDRT.

To ensure reliable accurate software, we will be testing the entire image processing application with focus being primarily on the image analysis and controller (part of the code that communicates between the GUI and image analysis module). The most critical section to test is

the image analysis as this is the end result JPL wants. Specifically, we will make sure that the analysis our code produces is with in 10% of JPL's hand done analysis for a dataset given to our team by the client.

2. Testing Framework

For testing our system, both unit testing and integration testing will be done using pytest. The two main frameworks we looked at were pythons unittest library and a 3rd party unit testing library, pytest. While unittest is built into python, we ultimately decided to use pytest for two reason. First it supports the python unittest library so, if need be, we can write unittest tests and only need one testing library. Second pytest has a robust set of features like automatic test discovery, and is still in active development. While it is another dependency, the benefits it provides to make testing easier out weight the additional dependency.

3. Unit Testing

Before we can test that all of our software modules are working together cohesively, we need to be sure that they work well individually. To make sure our software works correctly, we begin by testing the individual parts such as the control functions, utility functions, image analysis, configuration, and graphical user interface (GUI) functions. We have designed our software using the Model View Control architecture (MVC). This means that the Controller is set to handle the communication between the Model (image data) and the View (GUI), while the Model handles changes to its own data. We plan on testing each function for the GUI and image processing functions to make sure they are accessing and returning the correct data. However, it is important to focus on the image processing functions over the GUI, as they perform the operations on images that need to be tested and verified.

3.1 Image analysis (Model)

This module will be the primary focus for unit testing as all code contains logic that our team has written. It is also important for all functions in this module to be correct as they perform the primary dust analysis which is what J.P.L., our client, is after with this project.

Color Segmentation: Verify that color segmenting is generating correct output by passing in color (rgb) images to the function and viewing the output values. The output is a binary mask where white pixels are pixels that fall within a given color range defined in the script. Because there is no standard for dust analysis, the metric we are using are the results from JPL's hand drawn analysis.

Our client has provided the percentage of abrasions covered in dust for images of each rock type. We are using these to see exactly how close we are to the desired result. To improve accuracy of scripts that are not performing well, we will use Matlab's color segmentation application which lets users manually refine what colors make up the mask (ie manual iteration to improve output)

This is adequate for rock type E as the color of dust is distinctly different from the surface of the rock. Our results are already close to 10% of JPL's numbers. For other rock types, color segmentation is not advised as the rock surface and dust share some of the same colors.

Image Subtraction: This function subtracts the before image from the after image (after using gDRT). This will display pixels that have changed from the before image (eg blowing dust out of the picture.) However, this only works when the before and after images are aligned.

Testing this involves running the function on a series of before and multiple after images to make sure the right images are being subtracted (ie before - after != after - before).

Analyze Mask: This function takes in a binary image where white pixels are considered dust, and analyzes clustering of those pixels. In other words, given a region (eg 10 x 10 pixels), how many pixels in that region are dust. The result should mark the region using colors, where red = mostly covered, yellow = somewhat covered, green = mostly clear. To test this, we will use JPL's marked up images as a benchmark. Defining a region (10 x 10 or 15 x 15 pixels) and what constitutes "mostly covered" is up to our user. We will allow flexibility in how this function is run.

3.2 Graphical User Interface (View)

The expected output of the GUI is to pass along the configuration information (eg which rock type, what image analysis algorithm, etc) to the Control module. To test this, we can simply use assertions to check if the information is formatted properly. Also, testing should verify that the graphical items for the user interface are being displayed correctly in size, placement, and color. The majority of testing the GUI will be done in the integration tests, as there is little need to make testing environments for such a simple functionality. Since most of the functions in the GUI do not have boundary values to test against and will be tested more in usability and integration. We will focus primarily on the Control module and the Image module for unit testing.

3.3 Configuration

This is a basic class that is sent from the GUI to our Control object and determined how the Control should configure itself. As such many of the methods are simple getters and setters, having little to no logic, but making sure everything is set and working correctly is ideal so we will be testing these getters and setters.

Data Functions: The configuration file and class houses functions that return the desired data from the configuration sent from the graphical user interface. To test this we will call the functions in the class and compare them to the configuration that the user input.

3.4 Control (Controller)

While the Control object mainly leverages a pandas dataframe as its underlying data structure, there are some functions that contain custom logic that need to be tested. This testing ensures that the Control object is being instantiated correctly, is applying functions to the dataframe correctly, and is correctly saving off images that have been analyzed. As such, we will be testing the following functions:

Add Image Pairs: This function generates the image pairs for our data set and combines before and after images into the same row in the pandas dataframe. To make sure that the image pairs are being generated correctly we need to pass in a number of different image sets. The images are stored as "abrasion_030abraded.jpg" for example. This means that the image sets are to be generate by comparing the number and word following the number. To verify that the image pairs are correct we need to check if the rows contain matching pairs.

Run: As important as this function is it is simple to test. In order to view if it is working correctly we need to see if it returns a set of images that have been analyzed. We can also view if this is working if the analyzed image data is in the pandas dataframe.

Save: After a run has been completed we need to check the save function. Again this should be done by looking in the directory where the images were chosen from and see if the new image data is stored there after save was selected.

Apply Function: For this function we need to view if the pandas dataframe is applying the same image manipulation for all images in the set. What it's supposed to do is take a function and apply it to each item in a column of our choosing. To check this we need to return the image data found in the dataframe and verify that each image in the column has the same changes performed to them.

3. Integration Testing

Our system consists of three main modules, an Image module as the Model, a Control module as the Controller, and a tkinter Graphical User Interface (GUI) as the View. These are the main components of the Model, View, Controller architecture. Current the system has four lines of communication, where data is sent between modules:

- View to Controller
- Controller to Model
- Model to Controller
- Controller to View

As such we will be testing each transmission of data from one module to another to ensure that the modules are correctly integrated with one another and that data sent between modules results in the appropriate changes.

3.1 View to Controller

The GUI communicates with the Control module in a two step process. First, the View sends a configuration file to the Control. This object tells the Control how to create the underlying model objects, what functions to apply to those model objects, and the arguments to give to those functions when they are applied to the model objects. Next, the View tells the Control to actually run the functions on all of its underlying model objects. To test the communication from the GUI to the Control we will test it with 5 forms of a configuration object. This object contains four fields:

- Basepath
  - Path to image folder
- runList
  - List of functions being applied to model objects
- argList
  - List of arguments corresponding to functions
- Files
  - List of all files in the image folder
- rockType
  - Type of rock to be analyzed

Specifically we will test one configuration object with a bad Basepath where no images can be opened. Another with a bad runList that contains no functions. Another with a bad argList that is empty. Another with a bad File list that is empty. Another with a bad rockType, where no rockType is given. Finally, a configuration whose fields are all correct.

From the bad configuration files our system will fail to instantiate a Control object and tell the user that a bad configuration was given. For our one good configuration object, we can

expect that the object will be returned with all fields set and ready to be run. This will robustly test the first step of communication from the GUI to the Control.

The second step of this communication channel, should only work if a correct configuration was sent. Thus, to test the second step we will run the correct configuration from the first step, and expect no failures from the system.

3.2 Controller to Model and Model to Controller

The Controller communicates with the Model using control functions that are defined as part of the Control module. These functions take advantage of the underlying structure which is made up of rows, resembling a table from a relational database, but in memory, and runs the function with the row of the table as an input argument. Each control function then parses the necessary components from the row in the table and processes it. This process saves any changes made to the the Model objects in the Model objects themselves. Once a function has finished being applied by the Controller, all underlying Model objects will have been processed in some way. The control functions simply parse a given row for one of its fields, then calls a specific method from the Model object.

Integration testing between the Controller and the Model would simply be testing the logic of the 3rd party library (pandas) we are using that generates the Controllers underlying data structure. As such, it is not necessary to do integration testing on the communication between the Model and the Controller.

We have written very little custom logic to communicate between the two as the system leverages the underlying pandas dataframe to do the heavy lifting. Testing the communication between these modules would simply be testing that the logic in the pandas library is correct which is not something we need to test. Thus we will be ignoring integration testing between the Model and the Controller, and focusing on integration testing between the Controller and the View.

3.3 Controller to View

The communications from the Controller to the View will return two columns from the Controller. Both of which contain our own Model objects, or Image objects. This will be returned to the GUI after the Controllers run function has completed. This function runs the analysis on all Model objects, so on completion all Model objects will have had the necessary functions applied to them and contain the necessary data for the GUI to apply the various Views.

The only scenario that this will occur is when a correct configuration has been given to the Controller and the Controllers run function has been called and has finished execution. We then need to test the Views that the GUI provides to our users. Our client has only requested one View from the data that the Controller is passing back to the GUI. This view is of the image being analyzed and the final analysis of that image, along with a percentage for how covered that image is in dust.

To test this, we will simply run the system starting at the GUI, send a good configuration to the Controller and start the Controllers run function. Once that function finishes the GUI will attempt to create the desired view, in this case the display of the image, along with its analyzed

image and associated percentage covered. If the GUI is unable to create the View, this will indicate that something has happened along the Controller to View pipe as the other pipes would all be working for this test. If the GUI is able to create the View then the Controller to View pipe is working as intended.

4. Usability Testing

The purpose for our usability testing is to have some members from our client's business, JPL, test how they interact with our application. We want to know how our users will interact with the GUI to apply the various analysis methods that have been implemented so far for Rock Type E to images they give the application and receive the analyzed images. They will use the latest complete version of our prototype to conduct this portion of our testing plan.

Each user will do a one hour session to complete all the scenarios and answer the questions we will provide. This will give us feedback from a user perspective rather than a developer. In-between each scenario, the users are encouraged to write down any errors they encountered during the scenario and give feedback before moving on to the next one. The software will be run on desktops running MacOS High Sierra and Linux.

There will be three scenarios for each test session the user will conduct. The following scenarios are:

1. Select one pair of images to a folder of image pairs that would be analyzed;
2. Apply one to all of the image analysis methods that have been implemented to the images; and
3. View the results in the GUI.

They will repeat the scenarios with an increasing size of images in each batch since one of our requirements is being able to analyze multiple images at once. This will allow them to test to make sure this requirement has been met.

To measure the results for each scenario, they will take notes for any errors they encounter that caused them to not be able to finish the task, or they got an error but this error did not cause the task to be completed less efficiently. When they have completed all the scenarios, they will fill out a questionnaire to record their overall experience with our product. Some of the questions that will be in the survey include:

- How well does the application support your ability to accomplish key goals and tasks?
- What is your impression of the design of the GUI?
- Do you understand the content and does it help you accomplish your tasks? If not, explain why not.
- What are your overall impressions of the program? Does it adequately communicate what you can/are required to do with the application?
- Are there any recommendations you have? If yes, please explain.

We will also measure various quantitative metrics pertaining to how well the users are able to complete the tasks, how long they take to initially learn how to interact with the GUI, and how long it takes each user to complete each scenario. The first time they complete a scenario will be used as how long they learn. We want to take note of this time since most of the users will most likely not be familiar with Python, one of the languages we are using and the language

our GUI is written in. The times taken after the first one will help to see how they improve or not by repeating the same task using a different number of image pairs in each batch.

Other data points we will collect is what percentage of users were able to successfully complete each scenario and what percentage of users were able to complete the task without any critical/non-critical errors. These percentages along with the responses collected from the questionnaire will give us insight into what is not clicking for our users.