Automated Planetary Terrain Mapping of Mars Using Image Pattern Recognition

Test Plan
Version 2.0

Team Strata:

Sean Baquiro
Matthew Enright
Jorge Felix
Tsosie Schneider

Table of Contents

1 Introduction

Team Strata has agreed to work with client Ryan Anderson of the US Geological Survey Astrological Science Center (USGS), to create a computer program that would be beneficial to the research being conducted on Mars' surface. This document will address the details of the activities required to prepare for and support testing, details that define the sources of information used to prepare the test plan, different standards that will apply to the unit, integration and usability testing of the specified system, as well as define the various testing strategies and testing tools used to complete the testing life cycle of this project. The product being tested is a computer program that automates terrain mapping of Mars' surface using convolutional neural networks.

2 Unit Testing

The Automated Terrain Mapping of Mars project involved various functions implemented in Python and C++. The C++ functions were written using a C++ library called boost.python in order to enable interoperability between C++ and the Python programming language. This allowed us to use all of our functions through Python and only using C++ as the language to create Python functions that would process large JP2 and TIFF images quicker. With all the functions being used through one programming language, we were able to create a unit testing plan involving only Python. Python provides a variety of unit testing libraries including its standard testing module called PyUnit. PyUnit is a Python unit testing framework, essentially it is the Python language version of JUnit (Java testing framework). The unittest module supports test automation, aggregation of tests into collections, and independence of the tests from the reporting framework. Other important concepts unittest supports include test fixture, test cases, test suites, and a test runner. We will be able to create a collection of test cases and/or suites and use the test runner to execute the tests created and receive the outcome. Below is a detailed plan for unit testing our code in subsets of our system.

2.1 Unit Testing of Extension Plan:

The unit testing of the image processing module will consist of testing individual extension functions. These include:

- getDims: Extracts dimensional image metadata
- getImage: Populates a numpy array with pixel data

- getTraining: Populates a numpy array with pixel label information
- getLabel: Assigns a classification to a given numpy array
- writeImage: Writes a numpy image representation to disk

The functions serve as tools to obtain all the information necessary from the images and presents it in a constrained format required by the neural network architecture. As these functions make use of image files and numpy array's tests will make use of established python libraries for verification of certain aspects.

2.1.1 Test getDims

The setup for testing this function makes use of the python image library (PIL) in order to compare the dimensions returned by this function. This test makes use of an image created by PIL and opened as a numpy array. The shape of the PIL opened numpy is compared to the results returned by the getDims function.

2.1.2 Test getImage

This test uses a PIL created image, this image is then loading into a numpy array. A seperate numpy array with dimensions obtained from getDims filled with ones is then allocated. The numpy image containing only ones is then passed to the getImage function. The two numpy arrays are then compared element-wise for equality.

2.1.3 Test getTraining

The testing of this function will make use of a PIL created image. Since the function makes use of a bitwise XOR when an image is exclusively or'd with itself the result will be a numpy array of the given dimension where all elements are 0. The numpy passed to the getTraining function is then compared element-wise to a numpy XOR'd array for equality.

2.1.4 Test getLabel

The test setup of this function makes use of a 25600 by 25600 numpy array initialized with zeros. The array is then processed so 256 by 256 blocks along the diagonal are set to 255. A four depth nested loop is then used to test the getLabel function based on varying thresholds and image block dimensions.

2.1.5 Test writeImage

The setup of this test involves the creation of a numpy array with the dimensions 25600 by 25600. This numpy array is then written to disk. A second numpy array with the desired dimensions is passed to getImage function to load the pixel information that had been written to disk. The numpy used to write the image is then compared element-wise to the numpy populated with getImage for equality.

2.2 Unit Testing of Convolutional Neural Network Plan:

The unit testing of the convolutional neural network system will consist of testing individual functions. These include:

- get_labeled_data: Read input-array of the image and image labels and returns a list of tuples.
- trainNetwork: Begins training of the neural network on a test and training file
- makePredictions: Makes predictions on an input image and writes to an image which gets saved to the directory with marked predictions.
- getPredictionData: Creates the prediction data for the input file name taken in. Image must be in the same directory.
- loadDataset: Loads testing and training images into numpy arrays.
- convolutionalNeuralNetwork: Creates a convolutional neural network and takes the number of epochs as an argument.
- Download_image: downloads a hirise image when filename follow naming convention i.e., PSP_009650_1755_RED.JP2

The unit testing of the convolutional neural network will consist of testing individual methods and functions. These include get_labeled_data, trainNetwork, makePredictions, getPredictionData, loadDataset, convolutionalNeuralNetwork, and download_image. The functions and methods serve as components that are dependent on each other to make up the neural network behavior and architecture. It should be noted that running the tests will replace and get rid of any file named “net.pickle”, so all pretrained networks should be renamed or moved in a different location prior to running the tests.

2.2.1 Test get_labeled_data

Loads a sample test and training image and checks that the function returns the correct shape and type of the data and the correct number of labels for the particular set of images.

2.2.2 Test trainNetwork

Tests that the function creates a network with the sample test and train images. To be sure the function fully works, it creates the pretrained pickle data and checks if it exists in the system.

2.2.3 Test makePredictions

Creates a neural network instance of five epochs and loads the sample test and train images. After predictions are made, the output image dimensions are compared to the original image dimensions. The test passes if they are the same.

2.2.4 Test getPredictionData

Tests the function on the sample test image and creates image blocks of the default size 32. The output shape is then verified to be of size four (num images, channel, height, width). The dimensions of the output image are also compared to the original image.

2.2.5 Test loadDataset

Tests that the function loads the sample test and train images and verifies that the shape is equal to four dimensions and one dimension respectively.

2.2.6 Test convolutionalNeuralNetwork

This test function makes sure that the return type is of the Neural Network type by checking for its input layer and verifying that it is not none.

2.2.7 Test download_image

This test simply tests that the url is working and that a user can grab an image using the filename.

3 Integration Testing

The integration testing of our system will be broken up into three major modules and will help identify any incorrect exchanges of data between modules. The three major modules will include testing the pre-processing of a JP2/TIFF image into numpy arrays, testing the accuracy of the neural network, and testing the writing of a JP2/TIFF image using numpy arrays. The pre-processing and writing of a JP2/TIFF image will be tested by performing unit tests on the writeImage function. To train the accuracy of the convolutional neural network we will be using the K-fold cross validation technique. More information on the plan of conducting the cross validation technique can be found below in section 3.1.

3.1 10-Fold Cross Validation

As part of the integration testing, we will be using the 10-fold cross-validation technique to test the efficiency and accuracy of our convolutional neural network. Currently our input and output formats are large-scale JP2 and compressed TIFF images that the neural network uses to test and train on detecting terrain features. The Python machine learning library, Lasagne, contains its own cross-validation step when training and testing the convolutional neural network. The cross-validation step occurs during the training of the neural network. It splits the input data into 80 percent for training and then tests on the remaining 20 percent. This can been seen in figure 1 below. The table below is outputted during the training of the neural network and during each epoch we get the training loss which is the error on the training set of data, validation loss which is the error after running the validation set of data through the trained network, train/val which is the ratio between the two, validation accuracy, and the duration of each epoch. As the epochs increase, both validation and training error may drop. As the training error continues to drop, the network learns the data better and better. While this happens, if the validation error begins to rise, this is known as overfitting.

epoch	train loss	valid loss	train/val	valid acc	dur
1	6528.44945	0.57584	11337.18475	0.77497	75.41s
2	0.51446	0.57578	0.89350	0.77497	74.02s
3	0.51446	0.57574	0.89357	0.77497	75.22s
4	0.51447	0.57572	0.89361	0.77497	76.33s

Figure 1: Neural Network in Training

In addition to the cross-validation occurring during the training of the neural network, we will be conducting our own 10-fold cross-validation testing after the network has been trained. This technique is used to evaluate predictive models. In our case it will evaluate whether the convolutional neural network can or cannot detect a terrain type of interest in an image block of the overall image. The original image will be divided into ten equally sized sub datasets. Of the ten samples nine will be used to train the neural network and the remaining one will be used as the validation data to test the neural network. This procedure will then be repeated 10 times until every sub dataset is used for validation once.

One of the advantages of using this method is that every sub dataset gets used for testing exactly once and gets to be part of the training nine times. The only disadvantage of using the 10-fold cross validation technique is that the training of the convolutional neural network needs to be rerun ten times. This may be an issue since training the network once over 10 epochs can take about an hour. Using this method will potentially increase our testing time by ten, and even more so if the network should be trained over more epochs to improve accuracy. Using this technique will allow us to estimate the accuracy of our prediction model and allow us to detect and/or prevent overfitting. We will be able to detect overfitting by analyzing the predictions and seeing if we only get results through the use of one testing image. Prevention of overfitting will be done by training the convolutional neural network on ten different images instead on just one.

4 Usability Testing

4.1 Overview

The usability testing of our system will help confirm that it is ready for operational use and aid in identifying any problems that impact the ease of use, ease of learning, and overall satisfaction of the user. The scope of this testing will include evaluations

developed for the appropriate user of the command-line interface of the system. The results of the testing will be used to refine the system and command-line interface.

4.2 Target Audience

A crucial element of usability testing is selecting users with abilities and backgrounds that are representative of the system's intended typical user. For our project there is currently at this time only one intended user of the system, our sponsor Dr. Ryan Anderson. Therefore all usability evaluations will be designed for this specific user. Ideally, we would test at least five users to create a reasonable confidence in the results. Testing on a single user would normally be a risk, but since Dr. Anderson is the only user the system was designed for at this time, we can be confident in the results received from testing the system with this specific user.

4.3 Testing Process

Usability testing will be conducted on-site at the US Geological Survey Astrogeology Science Center (USGS) where Dr. Ryan Anderson's office is located. At the start of the evaluation a brief introduction and description of the usability test will be provided to the user. Then the user will be instructed to complete a list of tasks such as loading an image, training the network, and predicting on an input image, without any guidance using the command-line interface of the system and encouraged to think aloud and talk through their steps, problems if any, and opinions during the evaluation. The user's activity during evaluation will be monitored and documented by the test team in order to collect qualitative data.

Once the user has completed the tasks, an evaluation questionnaire will be provided to the user. Each task included in the task list completed by the user will have an associated question and evaluation scale. The user will answer the questions by circling the appropriate number on the scale and writing any additional comments in the "Feedback" section provided. In addition, the user will also be asked to verbally share any likes, dislikes, or suggestions with the testers. After the user has completed the questionnaire, a verbal interview will be completed if any problems occurred during the usability testing that need to be explored in more detail to find the answer to the following two questions:

1. What did the user find difficult about the task?
2. How could the task be improved or made easier?

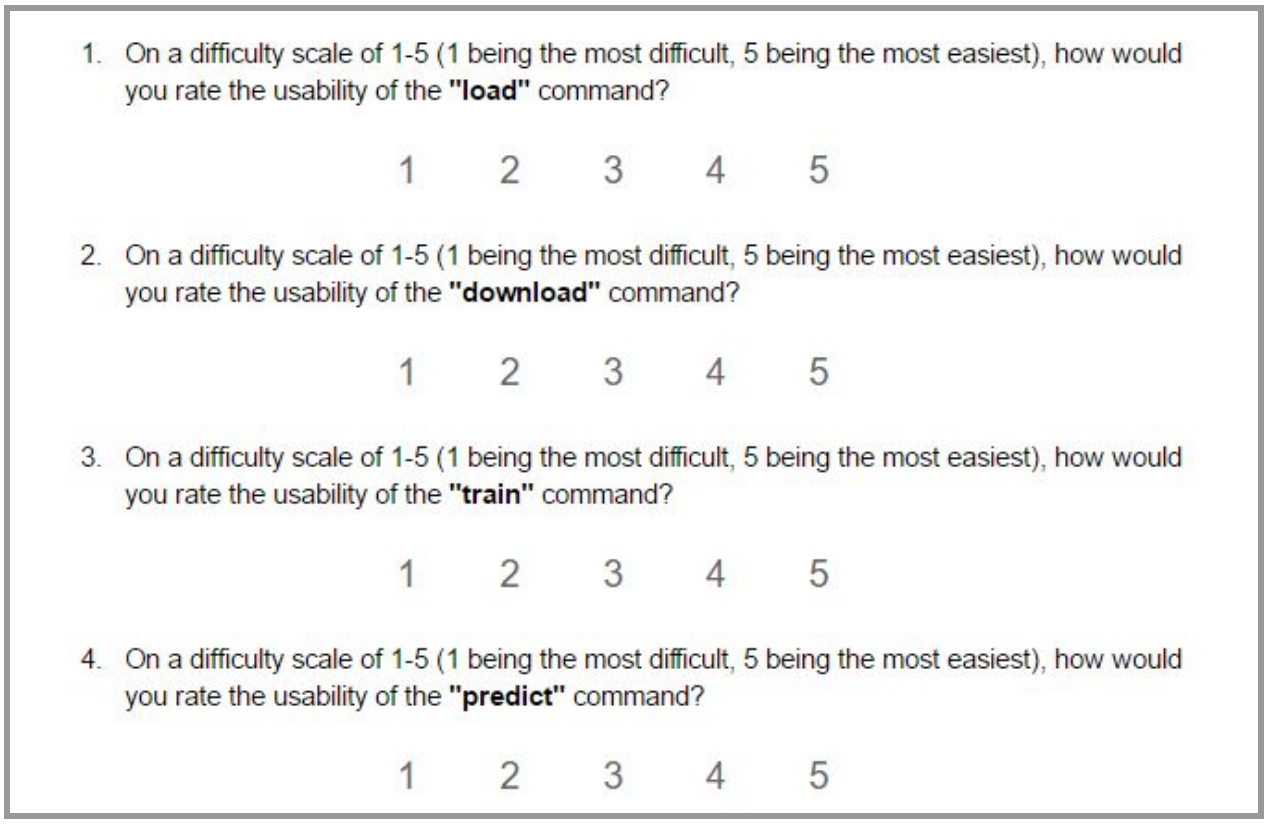

Figure 2: AToM Questionnaire example

4.4 Goals

Specific quantitative and qualitative usability goals were defined to evaluate areas of concern for testing. These goals will help guide the development of the tasks and evaluation questionnaire that will be used in the testing of the command-line interface of the system.

Goals defined:

- User will be able to successfully complete tasks without asking for assistance
- User will be able to successfully complete tasks with provided instructions
- User will be able to successfully complete tasks within a specific time-limit
- User will be able to successfully complete task on their first try
- User will feel that the time and effort required is appropriate for each task

- User will feel that task process is easy to understand and goes along with their own mental model of the process
- User will find the command-line interface simple and effective
- User will not feel that tasks require any unnecessary steps
- User will be satisfied with command-line interface and feel it provides all the functionality desired

4.5 Methodology

Usability will be measured based on a user's performance on a defined set of tasks using the command-line interface of the system. Data will be collected through the use of a "thinking out loud" process practiced by the user during evaluation while the testers take notes, as well as an evaluation questionnaire and verbal interview.

Metrics to be collected:

- Success rate (was the user able to complete the task?)
- Number of times user asked for help
- Number of positive/negative remarks made during evaluation
- Time required for a task
- Number and type of errors received during a task
- Overall Satisfaction with the system