APRIL 3, 2019

Sponsor: JASON ROBINSON OF BEAUTYMARK DESIGN STUDIO

Mentor: ISAAC SHAFFER

SOFTWARE TESTING PLAN

DIGITAL ROLL

Tyler Boice Caleb Johnson Tyler Malmon Brandon Measley

Table of Contents

1. Introduction

Tabletop Role Playing Games (TTRPGs) started in 1972 with a game called Dungeons and Dragons. What started as a small community of people and one game has blossomed into millions of players worldwide with many different games and versions, and today, there are an estimated 5.5 million people who play TTRPGs in the United States alone. These games have become so popular that they have spawned many web applications (e.g. Roll20, which has over 4 million registered users). Since TTPGs can be played online, players are not required to be in the same location anymore. This has led to the games becoming less personal, as players are either forced to leave their dice behind in order to use virtual dice rollers. They could still use their dice but since no one can see what they rolled it could lead to a state of distrust.

Our client, Jason Robinson of Beautymark Design Studio wants to create an application that solves this problem. His application with solve this issue by using object detection on a mobile phone to detect a player roll and send it to the group. Unfortunately, Mr. Robinson does not have the knowledge to create the models needed for object detection. It is team Digital Roll's goal to create a product that makes it easy for Mr. Robinson to collect the data needed for training and produce trained models that can be used on mobile devices.

The software we are creating is threefold: a workbench designed to create machine learning models that can be used to read dice on a mobile phone, a data collection application that will provide ease of use for collecting the massive amounts of data that machine learning requires, and an application program interface (API) that will make sure all the data is suitable for the workbench to use. These sections of our software will need significant testing to ensure that our product functions in the proper way we expect it to.

As unit testing is the process of testing small sections of code, it is incredibly important for finding any bugs that can occur within individual actions. For our software we are using unit tests in every major section of our software package. The data collection app will be tested to ensure that the

correct actions have been taken by each function. The focus of the API unit tests is to ensure that all API inputs and outputs are valid, and the required data is accurately stored and ready for the workbench. The workbench unit testing will focus on making sure it will be able to create models and that the user can load, verify, and save any preferences they might want to add. The testing will make sure that the models are created, accurate logs are created, and that the models can be converted to CoreML.

Our integration testing is going to focus mainly on the ways that the API and workbench function together. This is going to ensure the API gives proper data that the workbench needs to produce functional models, as well as making sure the model can be used with the API in a testing environment. The other piece of the workbench software that we are going to test is the model converter. This piece of the workbench needs to be capable of producing CoreML models that can be integrated nicely with an Apple application.

In order to make sure our software is usable; we need to test our products with users to ensure it is easily understood and easy to use. Since our client does not have extensive knowledge in machine learning, we will be testing with users in a similar background. For the tests we want to give the user as little information as possible to ensure that it can be used and understood with little explanation. This way when we hand the product to our client, he has everything he needs. The main pieces we intend to test are the data collection app and the workbench. The workbench is the most complicated part of our software; therefore, we will focus our usability testing more toward the workbench than the data collection application.

Our integration testing focuses mostly on the interactions within our software, making sure that the critical pieces of software are really meshing together in the ways that we want them to. We focus on the API because it is the piece of the puzzle that fits into almost every other aspect of our software. The workbench is secondarily focused here because it needs to work well with the API as well as converting the models it creates to CoreML so that the models work with the final expected implementation of our models. For usability, we are focusing on the two pieces of our software e that will be directly interacted

with by users. The data collection app needs to be understandable by users to the point that they can get work done with it in a timely manner that doesn't require much training. The workbench is the biggest part of the software that requires interaction, so making sure that it is understandable with the documentation provided is of key importance.

2. Unit Testing

In software engineering a unit is the smallest section of code that can be ran to perform some action. Often in programming units are any of the functions and methods created for the project. As functions are able to be run independently of each other and lend well to a modular structure, they can easily be executedone at a time with isolated parameters. This is what leads to unit testing. By selecting a particular function and giving it inputs, one can test the outputs in a highly controlled setting. This is useful for testing a functions actual outputs against its expected outputs. Additionally, it allows programmers to conduct negative testing by giving a function incorrect parameters to test the robustness of their code. Overall unit testing is an excellent tool for breaking down the components of one's code and help create more dynamic software. For these reasons we will be using unit testing on various methods and modules in our project.

2.1 Tools and Methods

There are many possible ways to facilitate unit testing for software. The method our team plans to use focuses on the two major coding languages we have implanted our project in Swift (in Xcode) and Python. For Swift we will be using the built-in unit tester for the Xcode IDE (Integrated Development Environment). This allows for direct access to all the functions in our app without needing to install other packages. Furthermore, it will allow for more detailed debugging due to the testing reports Xcode automatically generates. For Python we will use Pytest, which is a testing module that provides many basic functions to help manage unit testing. Both Xcode and Pytest provide methods which allow developers to create unit tests through a method of test and assert. The test is a function created

specifically to test many of the aspects of a single function, while the assert statements are within any given test function. Asserts work as a conditional statement that returns true if the function when ran matches your expected output and false if it does not match. With these two testing packages we plan to generate basic tests for all our necessary functions.

2.2 Unit Tests for Data Collection

The Data Collection we are creating will need to have many of their basic methods tested. This will ensure that they are generating the correct type of data before passing it off to our API.

2.2.1 Data Collection Units

Function Name	**Inputs**	**Outputs**	**Description**
getImageSampleBuffer	- Buffer: CMSampleBuffer	- UIImage ORnil	Captures an image and returns it
readAccelerometer	- none	- Float x - Float y - Float z	Reads and outputs the accelerometer data off the phone.
boundingBoxPressed	- Sender: Any	- none	Draws rectangle from two touchpoints
sharePressed	- Sender: Any	- none	Sends image, accelerometer and label data to API

Table 1: Data Collection Units

2.2.2 Data Collection Tests

Using Table 1, the following functions for the data collection units will be tested:

getImageSampleBuffer: We will be testing input values from the (Compressed Media) CMSampleBuffer to ensure that the media passed into the function is indeed an image and not some other form of media such as video, music, or audio. If the buffer contains an image the output should be an image and otherwise it will return nil.

readAccelerometer: We will be testing the outputs to ensure valid orientations of type float are always returned when accessing the phone's accelerometer.

boundingBoxPressed: We will be testing the range of inputs so the user cannot define a box with parts outside the image. Additionally, this function will test to ensure two positions have been assigned to be the corners of the bounding box.

sharePressed: We will be testing that an image has been taken, a snippet xyz accelerometer has been recorded, a bounding box has been drawn, and that labels for dice size and face are selected. This will ensure values are accurate for our API. Once the API returns a file, this function will check the file type and send it to an external folder.

2.3 Unit Tests for API

Function Name	Inputs	Outputs	Description
importer	- picture: file handle - xAccel: float - yAccel: float - zAccel: float - sizeLabel: int - faceLabel: int - bndBox: float array	- none	Handles input data passed to function. Converts data into strings before passing parameters to convert
convert	- image: string - xvec: string - yvec: string - zvec: string - size: string - face: string - bndbox: string	- none	Writes and formats the strings into a XML file.
output	- outputFile: file handler	- outputFile	Returns the generated XML file

Table 2: API Units

Using Table 2, the following functions for the API units will be tested:

Importer: We will be testing the inputs given to the function. All inputs should be within acceptable bounds. The picture should be of a jpeg format. The accelerometer variables should be in the valid range of -1.0 – 1.0 for all three inputs. The size label should be one of the seven possible dice sizes (4, 6, 8, 10(singles place), 10(tens place), 12, 20), and the

face label should be any number between 1 and the dice size. Lastly the bounding box should consist of four float numbers, with the first two numbers being the x and y coordinates of the first corner of the box and the second coordinates two being the coordinates of the second corner of the box.

Convert: We will be testing if a new blank xml file has been successfully initialized for this function. Furthermore, we will test to ensure the inputs are written into the file.

Output: We will be testing that the file we are outputting has all acceptable XML tags within it and that the file itself is not empty. This will ensure we are always handing off a valid XML file to our workbench.

2.4 Unit Tests for Workbench

Many of the functions we use in our workbench are based off TensorFlow and are already rigorously tested. Therefore, we have decided to primarily unit test the functions that we have specifically created or modified to better match the functionality of our workbench.

2.4.1 Workbench Units

Function Name	Inputs	Outputs	Description
load	- pref_path: filehandle	- none	Gets config file and sets up workbench based on preferences in file strings before passing parameters to convert
save	- save_path: file handle	- none (prints save information to console)	Saves an updated config file and changes old preferences.
run_export_tfserving	- weights: string - tiny: output: string - classes: string - image: int (img size) - num_classes: int	- none (creates and saves models)	Returns the generated XML file

export_coreml	- output: string	- none (saves CoreML model)	Converts a TensorFlow model into a CoreML model

Table 3: Workbench Units

2.4.2 Workbench Tests

Using Table 3, the following functions for the workbench units will be tested:

load: We will be testing to ensure a valid config file was passed to the workbench and we will test to ensure that the config file is formatted correctly. The testing of this function will ensure that users cannot pass an invalid config file that would prevent the process of creating models.

save: We will be testing that all preferences have indeed been updated. Additionally, we will test to ensure that if a file already exists with the same name as the new preference file, that the new file will not overwrite the old. Rather the new file will use an incremented number to ensure no data is lost.

run_export_tfserving: We will be testing this function to ensure proper logs are generated when this function runs. This function will be generating (You Only Look Once) yolo machine learning models.

export_coreml: We will be testing to ensure that a correct TensorFlow model is created with matching parameters to the type of information we collected in our API.

3. Integration Testing Modules

After we finish our unit tests and have verified every component of our product works, we must ensure individually tested pieces can function together as a whole; to do this we will use integration tests. Integration testing is a series of tests to expose defects in the interfaces of the software and the interaction between the integrated parts of the system. For our project there are many working parts that save data for other modules to read and utilize in their own process. As such, integration testing is necessary to

establish that the data is being saved correctly and can be used by different components of the program so it does not interfere with the expected operation. In the case of our project the main two components are our API and workbench which will need integration testing in order to see if these two major pieces can work together as a whole.

The API is designed to produce data that can directly be used as input into the workbench or for AI models. As such, the main focus of integration will involve that process. In addition to this the workbench itself features a CoreML converter module which can be tested independently and thus should also be tested for its ability to integrate.

3.1 Integration Testing Modules

3.1.1 API to Workbench

In order to verify the integration of the API and workbench the following is necessary. First the API should produce data that can be immediately used by the workbench without any extra conversion needed. Lastly the workbench should then be able to complete all unit tests involving data from the API. The method for testing this will require the API to produce a series of varied pieces of data so that this data can be placed somewhere the workbench can locate. The workbench will then be tested for full functionality on the data provided and the expected outcome will be a showcase of full functionality and a working AI model created from scratch via the data provided. A working AI model in this case will be one that can demonstrate having learned from the data provided.

3.1.2 API to Model from Workbench

The API data used for training the workbench should also be immediately usable for testing with any model produced by the workbench. The test for this would have the workbench produce a saved model and then it would test this model against various pieces of data that came directly from the API, so long as all data is fully usable and shows some amount of correlation to the given training then the test will be successful.

3.1.3 Workbench to CoreML converter:

The CoreML converter on its own should be capable of taking any valid saved model of *.pb* (TensorFlow model) or *.h5* (Keras model) format and converting it into an Apple CoreML model. Since the focus of the workbench is to output a saved. *pb* model, this is where integration testing comes in play. To fully test these two parts interaction the test would have the workbench produce valid *.pb* which then the CoreML converter will convert and output as a saved *.coreml* (Apple CoreML model) which will then be tested to make sure the conversion worked and training was transferred.

4. Usability Testing

After the integration testing is done and we have a functional product that meets our requirements, it is critically important that our client knows how to use it. Once we hand our product to our client, it is almost useless if he does not know how to use it. As developers it is easy for us to use and understand the product because we created it; therefore, we should not be the ones to determine the products ease-of-use.

Usability testing is the perfect way to test how easy our product is to use, because it gets the product in the hands of the end user so they can give feedback. The primary purpose for usability testing is to understand the perspective of the user, by having them use the product and so we can alter the product based on their feedback. This way we produce a product that we know is easy to use and understand.

Our product is divided into three portions: the data collection app, the workbench, and object detection app. Since the object detection application is there to prove the models created from the workbench are functional, it is not a priority that the app is easy to use. Instead we will just test its functionality with multiple models on multiple phones to verify that it's compatibility with other iPhones and iOS versions. For the other two portions of our product, the workbench and data collection app, we will be conducting in-depth usability testing. The usability tests that we conduct feature a member of our team acting as a proctor and a user that will test the product. For our tests we want to use users that have

the same knowledge as our client so we can make sure someone with his background can understand and user the product without us walking them through it. After considerable amounts of usability tests with users, we will conduct a final usability test with the client to verify its ease of use.

4.1 Data Collection App

Since the data collection app is simple, the testing for it will be basic. Our users will not need any pre-existing knowledge to use the app, so we will sample random users. For some of the users we will install the application onto their phone, while others we will have them test on a phone with the application pre-installed. The person giving the test will inform the user the purpose of the application and give them a document on how to use the it. Then we will supply a polyhedral dice and give the user no additional information. The user will then be instructed to vocalize what they are thinking as they use the app: Is it easy to use? Do they have any questions? Is anything unclear? If the user needs help, we will not assist them unless it is obvious, they will not figure it out. By not helping the user, we will be able see their thought process and determine why the issue is arising. The person giving the test will observe how the user uses the app: What do they attempt to click on? What do they take a while to figure out? Where do they struggle? After getting feedback and walking the user through the app, we can determine what issues and errors arise so we can fix them to ensure we make an app that our client will not have any problems with.

4.2 Workbench

One of the requirements described in our requirement specification document, is the user for the workbench is expected to have basic knowledge in file directories and command line experience but does not need any knowledge in machine learning. Therefore, the users we select will have basic experience with command line interfaces. The tests will be conducted over a communication program such as Discord or Zoom, so the user can screen share with the person conducting the test. Like the data collection app, the user will be told what the workbench does and be given a document containing all the

instructions needed for installation and usage. As they work their way through the document, the proctor of the test will ask the user to express what they are doing, thinking, and ask the same questions we asked for the data collection application. The workbench is incredibly complicated and has many components. Although all the components are laid out in the document, we realize it is easy to get confused. We will let the user struggle but will assist them if they seem stuck. It is crucial that the feedback is honest and that we get a wide variety of users so we can modify the document and workbench to be as simple to use as possible. These tests are also crucial because the installation process is long, and we have only conducted it on a handful of tests. Having the users test on their personal machine allows us to see if any errors or issues arise with certain hardware/software restrictions.

After we have tested on enough people that we feel we have an easy to use application, we will conduct one or two tests where the user walks through the entire workbench from data collection to object detection to ensure we have a seamless product. Finally, we will conduct a final usability test with our client to verify it's easy of use. We have been in constant contact with our client and he has confirmed that the interface of the comments of or product are good, but he has not yet used them. It is crucial to conduct a usability test on our client as this product is intended for him and he should know what to expect before the final turnover. We will also make sure that this test is conducted at least two weeks prior to delivery, so we have amble time to modify the product to his liking, and it is ready for product delivery.

5. Conclusion

At the end of the semester we will produce a product that will help our client create an application that improves the experience of millions of TTRPG's players. This product will allow our client to create polyhedral dice classifiers by using our data collection application, workbench, and object detection application. With these classifiers our client will have the ability to create an application that not only

allows online players to roll their own dice, buts also improves the overall experience help in-person players, such as players that are blind or players that want to integrate technology into TTRPG's.

In order to produce a product for our client that will be as error free and easy to use, we are going to be implementing several forms of testing: unit testing, integration testing, and usability testing. For unit testing we will test the major functions of the major components of our software. It is important to make sure that errors or fraudulent data is not making its way through our workbench, as well as ensuring that each section is creating and handling data in the correct way. For integration testing we will make sure our API works with both the workbench and the models the workbench creates, by verifying the data is usable by the workbench and testing the models. We will also ensure that the workbench works with the CoreML model converter so the default models that are created can be correctly converted for use with Apple products. Our usability testing will be focused on the two aspects of our product that require user interaction. We will verify that the data collection app is easy to understand and quick to use for users who have no experience with the app before. The workbench is the most important thing that we need to have tested for usability. As the core of our product, the workbench needs to be capable of being picked up by someone with little experience with command line interfaces and have them understand and use the workbench with only a helpful document. At the end of our testing we feel that our extensive efforts will produce a product that will be near error free and easy-to-use, as we are going to ensure the internal components function together, and the user side is compatible with users who have never seen our systems before.