Software Testing Plan

Version 1.0 - 11/8/2024

CRAFT (Ceramic Recording Automation and classiFication Team)

Project Sponsor:

Dr. Leszek Pawlowicz

Faculty Mentor:

Vahid Nikoonejad Fard

Course Organizer:

Dr. Isaac Shaffer

Team Members:

Kimberly Allison

Aadarsha Bastola

Alan Hakala

Nicholas Wiley

Table of Contents

Introduction

Deep learning models have proved to be a viable tool for improving resource utilization and classification consistency while cataloging ancient vessels. Team CRAFT, along with Dr. Pawlowicz, have worked to bring the power of deep learning models to archeologists through the development of functional software tools. The devised software solution contains three main components: a deep learning model for classification, a mobile application for field use of the model, and a conveyor belt system that allows for the use of the model in a laboratory setting.

As the development process reaches a near-complete state, a catalog of tests has been developed to ensure that a quality final product has been developed. These tests range from small, simple tests that check singular code units, to ones that encompass the user experience as a whole.

Our team plans on conducting three main types of tests: unit tests, integration tests, and usability tests. Unit tests, which are tests that check the functionality of the smallest units of code possible, will be conducted on each of the three main components of the program. This will ensure that the smaller functions that each program is built upon function as expected. Integration tests will be conducted to check for proper data transfer and associated error handling between modules. In this section, the critical operations between the deep learning model, the mobile application, and the conveyor belt system will be tested. Usability tests are the final type to be conducted on the software. In these tests, user interactions with the mobile application and conveyor belt user interface will be tested. This will ensure that the user-facing side of each application is designed in a way that is intuitive for users, easy to navigate, and responsive in simulated scenarios. The plan for usability tests also utilizes perspectives outside of those on the development team.

The developed testing plan includes the previously mentioned types of tests to ensure the testing coverage is both detailed and comprehensive. Program functionality will be validated to ensure that each piece of the software is functioning in the manner that the development team intended, while also identifying the presence of any weaker areas of the user interface. The testing plan outlined below will reveal further functional improvements to the application and allow for a streamlined user experience.

Unit Testing

Conveyor Belt Program Unit Testing

A unit test is a piece of code which intends to verify the accuracy of one part of your program. More specifically, when we do unit testing, we aim to divide our program up into its smallest individual units so that we can verify the correctness of every aspect in an isolated manner. The primary tool chosen to handle unit testing for the conveyor belt program is python's unittest unit testing framework. The creators of python have designed unittest to resemble JUnit, something previous classes have introduced us to making unittest a good choice for our background knowledge. Similar to JUnit, unittest supports most of its testing in an object-oriented manner.

The most frequently used tool form unittest used is the built-in class TestCase, which is the provided class whose methods allow you to perform unit tests. The unittest library provides an existing TestFixture class which allows you to handle the preparation, cleanup, and any other supporting work that is needed before performing a unit test. We will see later that a number of key unit tests for the conveyor belt program will require existing assets and variables to be prepared. While the TestFixture class would provide a solution, unittest provides an extension library unittest.mock which allows for the creation of mock objects which works better for our purposes.

An important tool from the mock objects library I regularly used when designing the unit tests was the patch() decorator, which handles patching attributes at an individual module or class level for our unit tests, more on how this was used later. Another essential tool from the mock objects library heavily utilized is the Mock class, specifically its subclass MagicMock. MagicMock objects allow us to easily make implementations of methods, data structures, variables etc… for the purposes of making assertions about how they act, and for providing dummy functions/variables that will react in a way that simulates their real code counterparts without the overhead of setting them up. Now that we know more about the tools used to design the conveyor program's unit tests, we can examine the tests.

Below, the in-depth plan for unit testing the conveyor belt program can be found. It includes descriptions of the unit tests processes and tools used and why specific situations are being tested. The tests are sorted by the high-level functionality that the code they are testing contributes to. The high-level concepts used to organize the unit tests also serve as the TestCase objects the individual unit test methods exist in.

1. Image Capture and Storage:

Before we begin the process of capturing images, we assume that there exists a valid place to store them. Due to this the program defaults to the directory used by the image model so that captured data is already in a place where it is ready to be used. The program will however allow users to specify their own directory to store the captured images. Our first test will check that our program will successfully create new directories, if the user provided the name of an existing directory to append to it will check that the directory exists, can be opened, and has no saving restrictions.

```
Test_directory_setup(self, mock_exists, mock_makedirs, mock_input)
```

is the test which handles this. This test uses the three mock objects in the parameter list to make assertions to check the previously listed steps this portion of the code handles. It also makes use of several patch fixtures which handle python's operation system library os's functions and returns. One example is

```
@patch("os.oath.exists", return_value=False)
```

To simulate the case where we attempt to find an existing directory path but fail. The method test_csv_setup employs a very similar process to test if our program successfully sets up a csv file, and test_exif_setup which handles asserting images stored exist with their properly stored geolocation data.

The main two tests which handle capture rather than storage are

```
Test_is_centered(self)
```

And

```
Test_background_format(self, mock_video_cap)
```

The first test evaluates the function is_centered() which handles calculating the moving object's position in relation to what is defined as the center of the screen. Since the output of this function is a boolean which tells us whether our object is centered this test is straightforward needing only two assertions to handle both cases.

The second test evaluates the program's ability to properly alter the camera feed, which it does to establish what the background is in relation to an object. Through its parameters it accepts a mock object which we can use to make assertions about how our altered video capture feed. It also makes use of a patch fixture to simulate an openCV video capture feed. This test process is to identify each alteration made to our raw video feed, then makes an assertion about a measurable difference that should be present if the alteration was made. An example of this is an assertion which, after a grayscale is applied, asserts that all pixels in the image should fall within a certain color range, and another that asserts the shape of the background follows that of a Gaussian blur.

2. Post Program Processes

After users conclude using the conveyor belt program there are a number of important post program processes that may or may not be performed depending on the users wishes. The following unit tests handle these important post program processes.

```
Test_cloud_upload(self, mock_init_app)
```

The unit test uses a MagicMock object to simulate the already existing firebase app and to simulate firebase credentials so that it can make assertions about what the program should do when there's valid and invalid credentials provided. It uses a similar mock object to stand in for the cloud and local database, the validity of this code portion can be easily checked by making assertions about what cloud data should exist given local data. This same cross referencing is seen in test_csv_append, which tests if the program properly appends its saved data to a user provided csv file.

The last conveyor belt program post run method to handle testing is the creation of TensorFlow datasets

```
Test_dataset_creation(self, mock_dir)
```

The structure of our TensorFlow dataset is represented by a patch fixture

```
@patch('keras.utils.image_dataset_from_directory')
```

This allows us to make assertions on the internals of our dataset, which will allow us to verify that we not only create a dataset but that it follows all our desired specifications. Using a combination of hard coded values to define things we know for sure about our dataset (such as our dataset does not contain a validation split, its interpolation, etc...) as well as variables to handle things like batch size and image size we can use assertions to verify a dataset is created with all it expected specifications.

Mobile App

Unit testing in Flutter allows us to verify the behavior of individual components in isolation. The primary tools used for testing in Flutter include the flutter_test package, which is built on Dart's test package, and mockito, which is used for mocking dependencies such as Firebase services or the TensorFlow Lite model. For this application, we focus on testing the core functionalities: Firebase authentication, model inference, image classification, and archival of classification data.

We used following tools and frameworks for unit texting in the flutter app:

- **`flutter_test`**: The core Flutter testing framework, which enables us to write unit and widget tests.
- **`mockito`**: A mocking library used to create mock objects for dependencies such as Firebase and TensorFlow Lite.
- **`test`**: Dart's core testing package, integrated within Flutter.
- **`firebase_auth_mocks`**: A package that mocks Firebase authentication functionalities for testing purposes.

This section outlines the key functionalities of the Tusayan White Ware classification app and the corresponding unit tests.

1. Firebase Authentication and Authorization

Firebase authentication is central to user access control in the app. The app requires authenticated users to classify the sherd images and store data.

Test Cases:

- **Test_authenticate_user**: This test verifies that users are correctly authenticated before accessing the classification functionality. Using mockito and firebase_auth_mocks, we mock Firebase authentication to simulate both successful and failed login attempts.

```
test('User login is successful', () async {
  final mockAuth = MockFirebaseAuth();
  final result = await mockAuth.signInWithEmailAndPassword(
      email: 'user@example.com', password: 'password');
  expect(result, isNotNull);
});

test('User login fails with wrong credentials', () async {
  final mockAuth = MockFirebaseAuth();
  try {
    await mockAuth.signInWithEmailAndPassword(
        email: 'wrong@example.com', password: 'wrongpassword');
  } catch (e) {
    expect(e, isA<FirebaseAuthException>());
  }
});
```

- **Test_register_user**: Similar to the login test, this ensures that new users are correctly registered in Firebase.

- **Test_user_logout**: Verifies that the user can log out successfully and the app handles this state correctly.

2. Image Classification Using TensorFlow Lite

The core of the app is its ability to classify images of Tusayan White Ware sherds using TensorFlow Lite. We mock the TensorFlow Lite inference process to isolate the classification logic from the model itself.

Test Cases:

- **Test_model_initialization**: This test ensures that the TensorFlow Lite model is correctly loaded into the app. We use mockito to mock the model loading process, verifying that the model file is accessed properly.

```
@GenerateMocks([Tflite])
test('TFLite model loads correctly', () async {
  final mockTflite = MockTflite();
  when(mockTflite.loadModel(
    model: 'assets/tflite/model.tflite',
    labels: 'assets/tflite/labels.txt',
  )).thenAnswer((_) async => 'Model Loaded');

  final result = await mockTflite.loadModel(
    model: 'assets/tflite/model.tflite',
    labels: 'assets/tflite/labels.txt',
  );
  expect(result, equals('Model Loaded'));
});
```

- **Test_image_classification**: This test checks that the app processes an image and returns a valid classification result. Using MagicMock, we simulate the TensorFlow Lite model inference process, returning a mock classification result for a given image.

```
test('Image classification returns expected result', () async {
  final mockTflite = MockTflite();
  when(mockTflite.runModelOnImage(path: anyNamed('path')))
      .thenAnswer((_) async => [
            {'label': 'Tusaya White Ware', 'confidence': 0.9}
          ]);

  final result = await mockTflite.runModelOnImage(path: 'test_image.jpg');
  expect(result[0]['label'], equals('Tusaya White Ware'));
  expect(result[0]['confidence'], greaterThan(0.8));
```

```
});
```

3. Image Archival and Data Storage with Firebase

After an image is classified, the classification result and the image need to be archived in Firebase. We use mock Firebase storage and Firestore instances to test these processes without interacting with the real Firebase backend.

Test Cases:

- **Test_image_upload**: This test ensures that classified images are correctly uploaded to Firebase Storage. We mock the Firebase Storage instance and check if the app attempts to upload the image file correctly.

```
test('Image uploads to Firebase Storage', () async {
  final mockStorage = MockFirebaseStorage();
  final ref = mockStorage.ref().child('images/test_image.jpg');

  when(ref.putFile(any)).thenAnswer((_) async => MockTaskSnapshot());

  final result = await ref.putFile(File('test_image.jpg'));
  expect(result, isA<TaskSnapshot>());
});
```

- **Test_classification_data_upload**: This test ensures that the classification result is correctly stored in Firestore. We mock Firestore and use assertions to verify that the correct data is written to the correct Firestore document.

```
test('Classification data is stored in Firestore', () async {
  final mockFirestore = MockFirestoreInstance();

  await mockFirestore.collection('classifications').add({
    'image_path': 'images/test_image.jpg',
    'label': 'Tusaya White Ware',
    'confidence': 0.9,
    'user_id': 'test_user_id'
  });

  final snapshot = await mockFirestore.collection('classifications').get();
  expect(snapshot.docs.length, equals(1));
  expect(snapshot.docs[0]['label'], equals('Tusaya White Ware'));
});
```

4. Post Classification Processes

Once the classification is complete, the user may want to view or edit their previous classifications. The app should retrieve these results from Firebase and display them to the user.

Test Cases:

- **Test_fetch_classifications**: This test mocks the Firestore collection and verifies that classification data can be fetched and displayed to the user.

```
test('Fetch classifications from Firestore', () async {
  final mockFirestore = MockFirestoreInstance();

  await mockFirestore.collection('classifications').add({
    'label': 'Tusaya White Ware',
    'confidence': 0.9
  });

  final snapshot = await mockFirestore.collection('classifications').get();
  expect(snapshot.docs.length, equals(1));
  expect(snapshot.docs[0]['label'], equals('Tusaya White Ware'));
});
```

The unit tests for the CRAFT mobile app focus on ensuring that each core functionality behaves as expected in isolation. The testing strategy involves mocking Firebase services, TensorFlow Lite operations, and other dependencies using mockito to keep the tests efficient and independent. The result is a suite of tests that ensure the app is robust and functions correctly, from user authentication to image classification and data archival.

Deep Learning Model

While the deep learning model itself cannot be tested like normal code, the dataset generation and loading can utilize unit testing to ensure proper functionality. Like the conveyor belt, we will be using the unittest framework included with Python for simplicity.

1. Reading CSV Files

The foundation of the dataset loading process is reading CSV files. CSV files are parsed to create a set of unique image filenames with associated labels. The test function below will be used to test reading CSV files.

```
Test_read_csv(read_csv)
```

The unittest takes in the read_csv function as an argument. Test_read_csv will generate a new CSV file and write it to file. Using the read_csv function, the new CSV file will be read into memory and tested against what was written to file, failing if there are any differences.

2. Creating Datasets

Using a dictionary filled with image filename and label pairs, the create_training_test_sets function splits this dictionary into a training and testing CSV file. Creating the test function below, the training and testing CSV files will be tested for correctness.

```
Test_create_training_test_sets(create_training_test_sets)
```

Integration Testing

Mobile App

Integration testing ensures that the various modules of a system work together by verifying interactions between them. While unit testing focuses on isolated functions, integration testing assesses how well modules communicate, ensuring data is correctly exchanged across system boundaries. For this mobile app, the integration points include the TensorFlow Lite (TFLite) model for image classification, Firebase for authentication and data storage, and the Flutter frontend that manages user interaction. The goal is to verify that data flows correctly between components and that the app can handle communication errors gracefully.

Goals of Integration Testing

The main goals for integration testing in this app are:

1. **Module Communication**: Validate that the app can successfully exchange data between Flutter UI, the TFLite model, and Firebase.
2. **Data Accuracy**: Ensure that the data, such as classification results and user information, is correctly passed, stored, and retrieved.
3. **Error Handling**: Ensure the app manages errors related to failed authentication, network issues, and TFLite model failures, providing clear feedback to the user.

Key Integration Points

1. Flutter UI and Firebase Authentication

- **Objective**: Verify that the app correctly manages user login and registration with Firebase.
- **Test Approach**:
    1. Simulate valid and invalid registration attempts, ensuring that errors are handled appropriately.
    2. Test login with valid credentials, checking that Firebase issues correct user tokens.
    3. Simulate token expiration and verify re-authentication handling.
- **Verification**: Confirm that the app processes valid logins, shows errors for failed attempts, and maintains authenticated sessions.

2. Flutter UI and TFLite Model for Classification

- **Objective**: Ensure the app communicates correctly with the TFLite model and that results are displayed accurately.
- **Test Approach**:
    1. Pass user-selected images to the TFLite model, verifying the classification output.
    2. Check if the classification results are properly displayed in the UI.
- **Verification**: Ensure the UI correctly displays the classification output, and results match the expected outcome.

3. Classification Data Storage and Firebase

- **Objective**: Validate that classification results are stored and retrieved correctly from Firebase.
- **Test Approach**:
    1. Simulate storing classification results, ensuring metadata like timestamps and user IDs are uploaded.
    2. Test retrieving data from Firebase and displaying it in the app's history section.
- **Verification**: Confirm successful storage and retrieval of classification data and ensure appropriate error handling for failed uploads.

4. Network Reliability

- **Objective**: Ensure the app handles network disruptions during authentication and data exchange.
- **Test Approach**:
    1. Simulate network loss during critical operations, such as data uploads or user authentication.
    2. Verify that the app displays appropriate messages and retries operations once the network is restored.
- **Verification**: Ensure smooth handling of network failures, with appropriate user feedback and retry mechanisms.

This integration testing plan for the mobile app focuses on key interactions between modules like Firebase and the TFLite model. The plan ensures proper data handling, effective error management, and that the app works as intended even under challenging conditions like network failures. Through rigorous testing of these boundaries, the app's robustness and user experience are safeguarded.

Conveyor Belt System

The main interacting modules of the conveyor belt system include the Firebase database, the TFLite deep learning model for classification, and the user interface for information display and settings manipulation. Integration testing for the conveyor belt system ensures that modules interact properly, maintaining the integrity of the data passed between them.

Integration Testing

The main goals for integration testing in this conveyor belt system are:

1. **Module Communication**: Ensure that data is passed between modules properly during program execution. Use of the model and local or cloud storage should correspond to selected user settings.
2. **Data Accuracy:** Validate that data passes between the modules in the correct manner, and integrity is preserved.
3. **Error Handling:** Ensure that when lines of communication between modules do fail, the integrity of the system is maintained, and the user is properly notified.

Key Integration Points

1. Conveyor Belt UI and Firebase Authentication

- **Objective**: Verify conveyor belt system's ability to interface with Firebase for user authentication
- **Test Approach**:
    1. Simulate the user login experience, ensuring that different login circumstances are handled appropriately.
    2. Test login with a set of valid credentials
- **Verification**: Validate that credentials can be both properly validated and created from the conveyor belt user interface. User and connection errors should be handled and communicated to the user.

2. Conveyor Belt UI and TFLite Classification

- **Objective**: Ensure that the conveyor belt UI properly passes images to the deep learning model
- **Test Approach**:
    1. Simulate situations where settings allow and disallow the classifications of images via the TFLite model.
- **Verification**: Ensure the proper classification information is retrieved from the model and stored appropriately. Any errors should be handled, and a corresponding error message should be properly displayed to the user.

3. Classification Storage

- **Objective**: Ensure that the conveyor belt UI properly uploads relevant session data to Firebase cloud storage.
- **Test Approach**:
    1. Simulate uploading directories of unclassified data to Firebase cloud storage.
    2. Simulate uploading directories of classified data to Firebase cloud storage.
- **Verification**: Ensure that data integrity is preserved as data is passed to Firebase, including user session information and associated sherd metadata. Errors should be handled, and relevant information should be displayed to the user.

The user testing plan for the conveyor belt system ensures that the system properly passes data between its main components. Testing ensures that when errors occur, the system is capable of responding in an appropriate manner.

Usability Testing

Mobile App

Usability testing evaluates how well users can interact with a system, focusing on whether the interface is intuitive and the workflow efficient. For the mobile app, the goal is to ensure that our client and development team can easily use the app to classify and store images of sherds. Since the app will be used by non-technical users, the testing will focus on ease of navigation, clarity of instructions, and overall user experience.

Usability Testing Plan

1. **Client Testing Sessions**
    - **Objective**: Ensure that the app meets the client's needs and is intuitive for them to use.
    - **Procedure**: The client will be asked to complete core tasks like uploading an image, classifying a sherd, and reviewing past classifications. Any issues they encounter or feedback they provide will be documented.
    - **Data Collection**: Observational notes, feedback forms, and error tracking during task execution.
2. **Team Testing Sessions**
    - **Objective**: Use the development team's perspective to uncover potential usability issues before client handoff.
    - **Procedure**: Team members will simulate user interactions, focusing on completing tasks efficiently and identifying any parts of the workflow that are unclear or cumbersome.
    - **Data Collection**: Internal feedback on navigation, task completion time, and interface responsiveness.

This approach ensures that the app is user-friendly for the client and addresses potential usability challenges before final deployment.

Conveyor Belt

The conveyor belt system is intended for use in a laboratory setting, with an emphasis on larger archival projects. The goal of the conveyor belt system is to improve the speed of the archival process, resulting in a focus on providing a system that is both responsive and intuitive. Usability tests for the program will prioritize the user experience navigating the program and seamlessly producing desired results.

Usability Testing Plan

1. **Client Testing Sessions**
    - **Objective**: Ensure that navigation and use of the system is intuitive and error-free
    - **Procedure**: The client will simulate a lab scenario in a laboratory environment. The client will be asked to complete a series of tasks, including creating a set of credentials, setting items on the conveyor belt for classification, and setting a different directory for stored images.
    - **Data Collection**: Observational notes, feedback forms, and error tracking during task execution.
3. **Team Testing Sessions**
    - **Objective**: Conduct testing sessions with the development team.
    - **Procedure**: The development team will simulate typical user scenarios with the conveyor belt system, checking for practicality of user interface choices and incorrect program behavior.
    - **Data Collection**: Time taken to complete tasks, user feedback, error tracking.

Comprehensive usability testing will ensure that the conveyor belt system provides a seamless user experience.

Conclusion

Through the use of our testing plan above, our team is confident that our final product will be shown to function correctly. Unit testing will prove the functionality of our code at a functional level showing that the individual parts work. Integration testing of the mobile app and conveyor belt system as planned will display proper connections between the individual parts of our project. Lastly, usability testing of our project as outlined in our usability testing plans will ensure that our final product is user friendly and simple.