COS301 Capstone Project

Team Syntactic Sugar

Jargon Sentiment Analysis Testing Policy

Team Members
Graeme Coetzee
Christiaan Nel
Ethan Lindeman
Kevin Coetzee
Herbert Magaya

Client
Compiax

July 19, 2019

Contat email: `syntacticsugar9@gmail.com`

Contents

1 Introduction

1.1 Purpose

This Testing Policy Document is intended to be a guide on the testing procedure, tools and practices that are followed when working on the Jargon Sentiment Analysis project. The remainder of this section describes the repository structure of the project. The remainder of this document then continues to describe the testing policies applied to the project.

2 Testing Process

2.1 Tests

The team makes use of a variety of different types of tests in order to fully test our application. These tests include but are not limited to:

- **Unit Tests:** are performed to ensure that individual functions and components of each system component performs correctly.
- **Integration Tests:** are performed to ensure that each subsystem correctly interacts with other subsystems.
- **Acceptance Tests:** are performed to ensure that the client is satisfied that we have met the requirements of their application.

2.2 Test Procedure

Jargon Sentiment Analysis leverages the full powers of Travis CI in order to utilize **Continuous Integration**.

Unit tests should be written before the tested function is written. The initial tested function shall be a stub that returns mock data in order to ensure the test is correctly written. Once the unit test has been written, we move on to writing and completing the function it tests. This is a test-driven development approach. Unit tests must be passed before merging to the master branch.

Integration tests should be passed before any code gets pushed to the master branch. Since we have a Microservices architecture, passing these tests is crucial to our system.

Acceptance tests are performed in a more unorthodox fashion, since we cannot regularly meet with our client due to their work responsibilities, we make use of the Demo sessions in order to communicate with our client and get their approval of the system requirements being met. We also schedule meetings every two weeks in order to communicate with our client.

2.3 Running Tests

All tests are automatically run on Travis CI whenever a branch is updated. Manual testing can be accomplished by executing the following commands from the project root directory.

```
$ cd tests
$ sudo chmod +x test.sh install.sh
$ ./install.sh
$ ./test.sh
```

The second command need only be run once, it is also linux specific and is required in order for the script to be executable. The `install.sh` need only be run once or can be run again to install missing development tools used during testing. A normal execution of tests would thus look as follows:

```
$ cd tests
$ ./test.sh
```

3 Updating and Adding New Tests

All unit-test related files for a service must be placed within the respective directory within the `services` directory. For example,

```
/
└── tests/
    └── services/
        └── controller/
```

would be the respective directory for the controller service.

The `test.sh` is a shell script which controls the testing procedure. It calls the respective test procedures for each service iteratively. If any single test fails then the entire procedure is deemed a failure. Since the services are language and framework independent of each other it simplifies the whole process of adding and modifying tests if there is a single common entrypoint script where each developer of a service/subsystem adds the necessary commands needed for their tests to be run by others.

Similarly the `install.sh` script must be modified to account for any tools required to run any new specific tests.

Details of the formatting is further documented within the script files themselves and is rather self-explanatory, intuitive and subject to change. Thus no further details are given here.

4 Testing Services

4.1 Cleaner

4.1.1 Description

The cleaner service cleans the tweet collection (removes retweets etc.) obtained from the`listeners` service.

4.1.2 Tools

The service makes use of the Mocha and Chai libraries in order to do unit and integration testing. These can be run by executing the following command within the `tests/services` directory within the console:

```
$ mocha test.js
```

4.1.3 Tests

The current unit tests exist to test the various endpoints of the Cleaner API used for cleaning data form different platforms. The endpoints currently tested is as follows:

- /twitter

4.2 Controller

4.2.1 Description

The controller service is responsible for controlling the flow of all service interactions. This is a back-end service for our system.

4.2.2 Tools

The service makes use of the Mocha and Chai libraries in order to do unit and integration testing. These can be run by executing the following command within the `tests/services` directory within the console:

```
$ mocha test.js
```

4.2.3 Tests

The current unit tests exist to test the various endpoints of the Controller API. The endpoints tested are as follows:

- /projects
- /projects/search
- /projects/delete
- /login

4.3 Flagger

4.3.1 Description

The Flagger service stores tweet data collected for projects in a database, the data will be used for training the neural network at a later stage.

4.3.2 Tools

The Controller subsystem makes use of the Mocha and Chai libraries in order to do unit and integration testing. These can be run by executing the following command within the `tests/services` directory within the console:

```
$ mocha test.js
```

4.3.3 Tests

The current unit tests exist to test the various endpoints of the flagger API. The endpoints tested are as follows:

- /flag/add
- /flag/train

4.4 Listener

4.4.1 Description

The listeners service retrieves phrases and sentences online source, such as Twitter.

4.4.2 Tools

The service makes use of the Mocha and Chai libraries in order to do unit and integration testing. These can be run by executing the following command within the `tests/services` directory within the console:

```
$ mocha test.js
```

4.4.3 Tests

The current unit tests exist to test the various endpoints of the Listener API used for aggregating live data form different platforms. The endpoints currently tested is as follows:

- /twitter

4.5 Neural Network Testing

4.5.1 Description

The Neural Network subsystem of our Jargon Sentiment Analysis project is the subsystem responsible for the actual analysis of tweets in order to determine the overall sentiment. It is a backend service for our system.

4.5.2 Tools

There were no specific tools we used to test the Neural Network subsystem.

4.5.3 Tests

The Neural Network subsystem cannot be thoroughly tested since it will always have varying values and we cannot tests a specific return value from the subsystem. Thus we do not have any specific tests written for this subsystem, instead we test it ourselves by comparing the results it returns to the results we expect it to return, within a range of error.

4.6 Web Service Testing

4.6.1 Description

The Web Application subsystem of our Jargon Sentiment Analysis project is the subsystem responsible for the client-side interacion with our system. It is essentially the ”front-end” of our system. This service runs client-side and sends requests to all our other services and displays results to the user.

4.6.2 Tools

For testing this service we make use of the standard Angular built-in tests. These can be run by executing the following command within the `services/web` directory within the console:

```
$ ng test <project>
```

4.6.3 Tests

The tests simply test whether the UI elements exist and were correctly created thus far. Further testing will be added as we expand our interface.