Silver Screen Testing & Validation Document

Final Version, December 5th 2016

Introduction

This document outlines the test plan for Silver Screen, a web application which allows users to access and track sentiments expressed through movie-related tweets. The architecture of the project can be broken down into four main parts: a Python framework used to collect and process data through external API calls, sentiment analysis modules which are used to score and analyse tweets, a database for storing sentiment scores and external data, and a web framework used to present information to the user and facilitate user requests. The following sections discuss various aspects of the test plan, in addition to providing justification for the test practices proposed.

Verification Strategy

As Silver Screen caters to a diverse audience, it was imperative that we reach out to a wide range of potential users when verifying that our web application's features and performance is representative of what our final user base (movie-goers and industry professionals) expect.

While the algorithmic base is critical to the success of our project, it serves little purpose if the data we collect is presented in an unclear manner. As a result, it was important that we aggregated user opinion regarding our site's UI to aid in creating an interface which is simple and clear, yet provides the detail which more advanced users expect. To accomplish this, we demonstrated Silver Screen to friends, family members, and instructors to help guide us in determining how data should ideally be presented. While most of these demonstrations were done on Silver Screen itself (especially for content-light pages such as the front and summary pages), GUI mockups for more complicated pages (such as content-heavy detail pages) were done in Clip Studio Paint and presented to our users digitally and in person. Demonstrations which were done on the live site were accompanied by a validation questionnaire (found on our GitHub repository), allowing us to acquire detailed responses from a wide variety of testers. By using this hybrid mockup/live-site approach, we were able to develop a good user interface while simultaneously developing deployable code.

Due to sentiment analysis' holistic nature, it is important for us to continue to assess the accuracy of our algorithms and ensure that the information provided on our site is reasonably accurate, even as we look past Silver Screen's release. This is especially critical due to the fact that we display sentiment scores attached to specific tweets, as tagging a tweet with an incorrect score could greatly diminish the user's trust in our system.

Non-functional Testing and Results

The following test suite aims to assess more holistic, non-functional aspects of our program, with a focus on system stability and maintainability.

Test #	Requirement Purpose	Action/Input	Expected Result	Actual Result	P/F
NF1	Endurance Test	Run the site for a week without restarting or other intervention	No changes to user experience of access times result	No changes to user experience of access times result	P
NF2	Cross Platform Consistency Test	Search for the same movie using different platforms and browsers (including mobile)	Data displayed is identical in content and description for PCs, data is neatly re-formatted and responsive for mobile	Data displayed is identical in content and description for PCs, some data is neatly re-formatted and responsive for mobile	P
NF3[1]	Sentiment Analysis Accuracy	Assign scores to a series of tweets by hand and compare with algorithmic scores	Scores created through human assessment match the scores algorithmically generated	Scores created through human assessment mostly match the scores algorithmically generated	P

Sentiment Analysis Algorithm

To improve the scores produced by our sentiment analysis algorithm we regularly re-evaluated results and tweaked our process as we progressed through the creation of Silver Screen. In addition to making changes based on educated guesses and intuition, we created a test suite to compare the results of our sentiment analysis to the results we expected to see. To accomplish this we manually retrieved and classified a set of tweets to create the expected set of sentiment scores. We then ran the same tweets through our sentiment analysis algorithm and compared how similar our results are, allowing us to gradually tweak our parameters to obtain better, more human-like scores.

Functional Testing Strategy

Testing Strategy

In order to test our application at a low level we utilized Python/Django's *unittest* module to implement and execute unit tests. To ensure that unit tests were written, executed, and updated frequently, we took a two-layered approach. First, each team member wrote test cases for the modules they were responsible for writing, preferably before the code was

[1] See *Sentiment Analysis Algorithm* Section

written. As they progressed through their implementation they regularly ran these tests, added additional tests, and implemented bugfixes accordingly. This regression/test-first approach allowed developers to continually reflect on their code base and anticipate future difficulties, significantly reducing the number of latent/recurring bugs. When the developer was finished with their implementation, they issued a pull request from their branch (containing their code and testbenches) to the ‘dev’ branch. Team members were then expected to assess the completeness of the code within the pull request, and provide feedback to the developer to assist in making corrective changes (if required). Once multiple developers positively assessed the pull request and its corresponding test suite, the pull request was approved and the ‘dev’ branch was updated accordingly. Additional testing was performed on the dev branch at regular intervals by all team members - especially during the end of each sprint. Once we were confident that there were no outstanding issues with the codebase in dev, we pulled dev into master (our live site’s deployment branch).

In addition to utilizing unit testing, we performed higher-level integration and system testing through the use of Travis CI, a popular continuous integration suite which is feature-rich and compatible with both GitHub and Heroku. We configured Travis to run our tests (collectively written between the developers) every time we pushed to any branch or issued a pull request. This functionality was especially useful during merges, as integration bugs became immediately apparent upon automated builds. Travis was also configured such that the results of our testing appeared in the README of our Github repo, allowing quick confirmation that the build was currently sound.

Bug Tracking

We utilized GitHub’s built in issue-tracking system to help manage outstanding bugs and track the code-completeness of our program. We chose GitHub’s issue tracking system over alternative bug-tracking systems (such as Bugzilla or Jira) due to its integration with GitHub and the other development tools we are currently using (including Slack).

Our bug classification scheme is as follows:

Critical - Bugs which prevent the program (or critical portions of it) from functioning or outputting a sentiment score in any case, restricting development until it is fixed.
Severe - Bugs which cause the output of our program to be significantly incorrect or non-existent, but are limited to a specific segment of inputs (i.e. non-existent movies).
Warning - Bugs which cause the output of our program to be incorrect for specific inputs, but do not otherwise impede the customer’s experience.
TODO - Issues within our program which do not cause our program’s output to be inherently incorrect, but could be subject to improvement. These issues are generally restricted to bugs which would not be necessarily noted by the user without close inspection, and hence can be included in public releases depending on our release schedule.

We used the following Github issue flags to define a bug's status:

Open - Any bug that has not yet been resolved.
Close, flag with devtest - A bug that has been resolved and has a corresponding unit test, but has not yet been tested within the development branch.
Close - A bug that has been resolved, has a corresponding unit test, and has been tested within the development branch.

Adequacy Criterion

As we strove to have an established threshold which we can use to deem our testing process acceptable for our purposes, we set a few goals to determine the completeness of our testing, as follows:

- All modules critical to functionality must be covered by at least one test
- Methods which rely on external resources (such as API requests) must be tested under normal and error-state operating conditions
- All use cases and requirements must be covered under both unit and continuous integration tests

In general, our goals aimed to ensure that the core performance of our system was always as expected and to ensure that the use cases can always be performed - especially in the scenario where failures in our API calls result in us not being able to serve data appropriately.

Test Cases and Results

Our current functional test case suite is broken up into the following subsections, as follows:

API Tests

Test #	Requirement/ Purpose	Action/Input	Expected Result	Actual Result	P / F
F1	TwitterAPI.search_movie() with invalid movie	Call TwitterAPI.search_movie(movie) with an object that is not of type Movie	TwitterAPI returns None	TwitterAPI returns None	P
F2	TwitterAPI.search_movie() with movie with an empty title	Call TwitterAPI.search_movie(movie) with an object of type Movie that has a title that is not a string of size > 0	TwitterAPI returns None	TwitterAPI returns None	P
F3	TwitterAPI.search_movie() with valid movie	Call TwitterAPI.search_movie(movie) with an object of type Movie that is valid	TwitterAPI returns List<twitter.Status>	TwitterAPI returns List<twitter.Status>	P
F4	TwitterAPI.search_movie() with complex title	TwitterAPI.search_movie(movie) with an object of type Movie that has a valid long and complex movie title	TwitterAPI returns List<twitter.Status>	TwitterAPI returns List<twitter.Status>	P
F5	Invalid Movie Submission Test	A user requests information about a movie that does not exist in the OMDB database	User is redirected to the form page and sees “Sorry, we couldn't find a move with that title.”	User is redirected to the form page and sees “Sorry, we couldn't find a move with that title.”	P

Sentiment Analysis Tests

Test #	Requirement Purpose	Action/Input	Expected Result	Actual Result	P / F
F6	Check Sentiment Analysis polarity_scorer	Create various word sequences to check whether score modification of preceding words is working correctly	Preceding words have their score modified appropriately	Preceding words have their score modified appropriately	P
F7	Test sentiment modifiers	Test sentiment containing the words “but” and “at least” in the middle of the sentence (both upper and lower cases)	Sentiment score is altered appropriately when these words appear followed by a phrase with opposing sentiment	Sentiment score is altered appropriately when these words appear followed by a phrase with opposing sentiment	P
F8	Test capitalization boost on sentiment score	Get tweet sentiment for sentences with different levels of capitalization	Sentiment intensity is increased appropriately by capitalization of charged words	Sentiment intensity is increased appropriately by capitalization of charged words	P
F9	Test words and symbols tokenizer	Get tweet sentiment for sentences with misspelled or non emotive words, symbols, and numbers	Sentiment intensity is unaffected by misspelled or non emotive words, symbols, and numbers	Sentiment intensity is unaffected by misspelled or non emotive words, symbols, and numbers	P
F10	Test sentiment normalization	Pass a variety of sentiment scores within and outside the normal range	All values returned should be within the normal range of sentiment scores. Higher values should return results in the upper part of the normal range, likewise for low values.	All values returned are within the normal range of sentiment scores. Higher values return results in the upper part of the normal range, likewise for low values.	P

Data Container Tests

Test #	Requirement Purpose	Action/Input	Expected Result	Actual Result	P / F
F11	Unique Tweet Constraint	Create new Tweet that is already in the database	There should be no duplicate tweet IDs in the tweets table, constructor returns existing tweet in DB	There are no duplicate tweets IDs in the tweets table, constructor returns existing tweet in DB	P
F12	Tweet Object Invariant Test #1	Call Tweet().fillWithStatusObject(data) with data that is not of type twitter.Status.	Tweet object should return itself in the same state as it was before the method call was made.	Tweet object returns itself in the same state as it was before the method call was made.	P
F13	Tweet Object Invariant Test #2	Call Tweet().fillWithStatusObject(data) with data that is of type twitter.Status.	Tweet object should return itself. Data in Tweet object should match that of the data object passed in the method call.	Tweet object returns itself. Data in Tweet object matches that of the data object passed in the method call.	P
F14	Movie Object Invariant Test #1	Call Movie().fillWithJsonObject(data) with data that is invalid or incorrectly formatted.	Movie object should return itself in the same state as it was before the method call was made.	Movie object returns itself in the same state as it was before the method call was made.	P
F15	Movie Object Invariant Test #2	Call Movie().fillWithJsonObject(data) with data which is valid	Movie object should return itself. Data in Movie object should match that of the data object passed in the method call.	Movie object returns itself. Data in Movie object matches that of the data object passed in the method call.	P

HTTP Response Tests

Test #	Requirement Purpose	Action/Input	Expected Result	Actual Result	P/F	Notes
F16	Correct GET results response	Issue GET request to '/results' with valid query	Server returns HTTP-200 with 'form' parameter of type Form	Server returns HTTP-200 with 'form' parameter of type Form	P	
F17	Correct POST results response	Issue POST request to '/results' with valid query	Server returns HTTP-200 with 'query' parameter equal to user query	Server returns HTTP-200 with 'query' parameter equal to user query	P	See below
F18	Correct results response on invalid request method	Issue PATCH request to '/results'	Server returns HTTP-403 METHOD NOT ALLOWED	Server returns HTTP-403 METHOD NOT ALLOWED	P	
F19	Correct GET index response	Issue GET request to '/index' with valid query	Server returns HTTP-200 with 'form' parameter of type Form	Server returns HTTP-200 with 'form' parameter of type Form	P	
F20	Correct POST index response	Issue POST request to '/index' with valid query	Server returns HTTP-200 with 'form' parameter of type Form	Server returns HTTP-200 with 'form' parameter of type Form	P	
F21	Correct index response on invalid request method	Issue PATCH request to '/index'	Server returns HTTP-403 METHOD NOT ALLOWED	Server returns HTTP-403 METHOD NOT ALLOWED	P	
F22	Correct GET about response	Issue GET request to '/about'	Server returns HTTP-200	Server returns HTTP-200	P	
F23	Correct GET overview response	Issue GET request to '/overview'	Server returns HTTP-200	Server returns HTTP-200	P	

*F17: This behaviour may seem strange but it is correct and has to do with the way we handle requests using AJAX for better UX.