

# Silver Screen Testing Plan

Initial Version, October 28th 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 via external API calls, sentiment analysis modules which will be used to score and analyse tweets, a database for storing sentiment scores and external data, and a web framework to present information to the user and facilitate user requests. The following sections will discuss various aspects of the test plan, in addition to giving justification for the test practices proposed.

## Verification Strategy

As Silver Screen caters to a diverse audience, it is imperative that we reach out to a wide range of potential users when verifying that our web application's features and performance are representative of what our final user base will 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 is important that we aggregate user opinion regarding our site's UI to help create an interface which is simple and clear, yet provides the detail which more advanced users expect. To accomplish this, we plan to initially demonstrate 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 will be 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) will be done in Photoshop or a similar editor and presented to our users digitally or on paper. By using this hybrid approach, we hope to develop a good user interface while simultaneously developing deployable code.

As we move into the beta stage of the project, we aim to expand the reach of our test-base by transitioning to feedback acquired solely through online interaction. Through use of social media we hope to acquire feedback from a more diverse user base, in addition to allowing us to test our implementations on a live site. Feedback on the site's usability will be collected via an online form (likely a Google poll, but potentially collected on Silver Screen itself) and will be assessed at regular intervals in the beta stage. If we find that we are not receiving enough feedback through this polling scheme, we may attempt to incentivise feedback through a giveaway or other reward structure.

Looking past the beta stage, it is also important for us to assess the accuracy of our algorithms and ensure that the information provided on our site is reasonably accurate. This is especially critical if we opt to 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 <sup>1</sup>	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 will continue to re-evaluate results and tweak our process as we progress through the creation of Silver Screen. In addition to making changes based on educated guesses and intuition, we plan to create a test set to compare the results of our sentiment analysis to the results we expect to see. To accomplish this we will manually retrieve and classify a set of tweets to create the expected set of sentiment scores. We will then run the same tweets through our sentiment analysis algorithm and compare 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 plan on utilizing Python/Django's *unittest* module to implement and execute unit tests. To ensure that these tests are comprehensive, we will be tracking our test coverage with Python's *coverage.py* module. This module tracks all unit tests we have written and gives statistics on the code coverage of each test in

---

<sup>1</sup> See next page for further details

addition to total code coverage. This will allow us to identify problem areas which require more testing.

To ensure that unit tests are written, executed, and updated frequently, we will take a two-layered approach to unit testing. First, each team member will write test cases for the modules they are responsible for writing, preferably before the code is written. As they progress through their implementation they will regularly run these tests, add additional tests, and implement bugfixes accordingly. When the developer is finished with their implementation, they are to merge their branch (containing their code and testbenches) with the 'dev' branch, and run the entire suite of tests in dev. Once multiple developers have assessed the tests and corresponding code on the dev branch at the end of each sprint, we will merge dev with master (our deploy branch).

In addition to utilizing unit testing, we will perform 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 will configure Travis to run our tests (collectively written between the developers) every time we push and issue pull requests, generally twice per product iteration. We will also set up Travis so the results of our testing appear in the README of our Github repo. Using Travis to automatically test our system when we merge into master will allow us to ensure that our system is working correctly in addition to managing our code's deployment to Heroku.

## Bug Tracking

We plan on utilizing GitHub's built in bug-tracking system to help manage outstanding issues and track the code-completeness of our program. We have chosen 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 will use 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 wish to have an established threshold that which we can use to deem our testing process acceptable for our purposes, we have set a few goals to determine the completeness of our testing, as follows:

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

In general, our goals aim to ensure that the core performance of our system is always as expected - especially in the cases where failures in our API calls result in us not being able to serve data.

## 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 movie with that title."	User is redirected to the form page and sees "Sorry, we couldn't find a movie 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 for non emotive words, symbols, and numbers	Sentiment intensity is unaffected by misspelled for non emotive words, symbols, and numbers	Sentiment intensity is unaffected by misspelled for 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
F10	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
F11	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
F12	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
F13	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
F14	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
F15	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	
F16	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
F17	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	
F18	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	
F19	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	
F20	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	
F21	Correct GET about response	Issue GET request to '/about'	Server returns HTTP-200	Server returns HTTP-200	P	
F22	Correct GET overview response	Issue GET request to '/overview'	Server returns HTTP-200	Server returns HTTP-200	P	

\*F16: This behaviour may seem strange but it is correct and has to do with the way we handle requests using AJAX for better UX.