Design Document

Silver Screen

November 28th, 2016

Introduction

This document outlines the design and architecture of Silver Screen - a web application which movie-goers can utilize to assess and track sentiments expressed within relevant movie-related tweets over time. Within each of the following sections we will share the rationale behind our design decisions and provide insight into the specific platforms and structures we have chosen for this project.

Developer's Note

This document overviews higher-level design rationale, rather than developer specific implementation information. Developer specific documentation (including project deployment instructions) can be found in Silver Screen's README, found here.

System Architecture and Rationale

While users solely interface with the web front-end of our application, a large majority of our development focus was on the acquisition and processing of data in preparation for the front-end presentation. As such, we broke down our application as follows:

1. A base Python framework used to facilitate external requests and data transfer between modules
2. Internal sentiment analysis modules which use natural language toolkits to break down and analyze tweets based on their sentiment
3. A database which stores sentiment scores, external scores (ex. Rotten Tomatoes, IMDB), and other movie information for historical lookup
4. A user interface which presents the user with sentiment scores and facilitates user requests for specific movies or historical data

Python

The bulk of our backend data processing is implemented in Python. Python was chosen because of its many well-known packages for performing data processing. Additionally, using only Python for the internal data processing reduced the complexity of interfacing modules with one another.

To acquire third-party data about movies, we utilized three API packages: `omdb`, `twitter`, and `imdb-pie`.

OMDb

The OMDb API library is the portion of our code which acts as our link to IMDB, the Internet Movie Database. The use of the OMDb library is two-fold. First, it acts as our search engine, relaying the user's query to IMDb to retrieve a list of movies matching the query (performed by calling the `omdb.search_movie()` method). This search step can sometimes be avoided through the use of caching in our database (if we have seen the movie before we don't have to look it up on IMDB as we have all of the pertinent information stored already). Once we have decided which movie we wish to display (the item with the most IMDB ratings in the search result list) we perform a query to pull additional information (such as box office results, poster images, etc) using the `omdb.request()` method. OMDb also provides us with Rotten Tomatoes information, including user and critic review scores.

Twitter

The Twitter API is arguably the most critical library within our project. After we have retrieved the movie information from OMDb (or from our local database, in the case of caching) we proceed to search for relevant tweets using the `twitter.api.GetSearch()` method. It is important to note that we perform two filtering steps during this search. First, as Twitter counts retweets as individual tweets and we don't want to factor in the same tweet multiple times, we ignore all of a tweets' retweets (but we weight the tweet's sentiment score based on its number of retweets and favorites). Secondly, we ignore all tweets with links, as we have found that these tweets are generally from corporate accounts and are thus not representative of the population we are attempting to model.

IMDB-Pie

This library is used to supplement the OMDb library's functionality in retrieving movie information. While we used OMDb for the vast majority of our movie information retrieval due to the large amount of data it returns for a single movie, it is unfortunately lacking in the ability to provide a list of popular movies - severely restricting our ability to implement functionality which didn't depend on direct user input (such as the random movie function). By utilizing IMDB-Pie's functionality (most notably, its `top_250()` function), we were able to acquire popularity information, which we then use in subsequent OMDb lookups.

Natural Language Toolkit (NLTK)

Originally, we had planned to make extensive use of Python's NLTK package in writing our sentiment analysis algorithm. As we came to understand what was necessary for the algorithm, however, the way in which we used NLTK transformed.

The most useful application for NLTK was its ability to analyze various corpora of words and their sentiment scores. We were able to use NLTK with SentiWordNet to compile a lexicon of

words and ascribe sentiment scores to them. We used several other publicly available sentiment lexicons, using NLTK to analyse them and combine all of these lexicons to create a larger, more accurate lexicon.

Aside from its ability to analyze corpora, NLTK also offers more powerful text file parsing capabilities compared to the parser built into Python. We used this “load” feature in order to read our lexicon text file into a dictionary object which the algorithm then uses to determine whether it should ascribe a score to a given word. The NLTK package was extremely beneficial because it allowed us to do this in far fewer lines of code (and significantly faster) than Python’s normal file I/O system.

Sentiment Analysis

Our sentiment analysis algorithm was originally composed of two large parts - the algorithm itself and a pre-built lexicon of words and their various attributes. Emphasis was put on building the lexicon first, as the testing and construction of the algorithm was contingent on at least a somewhat functional lexicon.

By using WordNet’s synset system (which provides a set of synonyms for each word), we broadened our lexicon by assuming that synset words have similar positive and negative sentiment. NLTK also allowed us to make use of the SentiWordNet lexicon, which is a subset of WordNet that ascribes positivity, negativity, and objectivity scores to all of its words. There are a variety of other lexicons which specialize in parsing the meaning of texts from social media, which we explored in our effort to create an expansive and accurate lexicon.

Within the realm of sentiment analysis, there seem to be two major archetypes in how to quantify the meanings of words, a polar system or an intensity system. The polar system simply scores a word on whether it is positive or negative, based on its context. The benefits of this system are that it offers an easy path to create a massive lexicon which is able to pull some meaning from a given sentence. The more difficult method is to create a system which measures the intensity of the positivity, negativity, and neutrality of a given word, so that we can paint a more accurate picture of what a tweet is actually trying to say.

In the end, we decided to use the second method. By aggregating the sentiment scores for words found in several different lexicons and averaging those scores, we were able to assemble an accurate lexicon which was able to pull a good deal of meaning, both negative, positive and neutral from a tweet. This holistic approach was more difficult not only in the construction of the lexicon, but also in the algorithm itself. We needed several modules which checked for the presence of not only individual words but punctuation, capitalization, and phrases/idioms as well. In doing so, our algorithm had fewer blind-spots than similar algorithms often have, and was better able to determine the actual sentiment of complicated tweets which use negating words like ‘don’t’ or ‘wasn’t’ etc.

Once we completed the lexical model, we began to build our algorithm in a modular way. By using some of the patterns we noticed in constructing the lexicon, as well as trends we noticed in the tweets we were pulling with our Twitter API, we learned about the various adjustments our algorithm would need to make in order to more accurately score a tweet. Unit and regression testing were also key to the algorithm's development, not only allowing us to ensure that our code was robust, but also providing us information which we could use to develop additional modules to fine-tune the system.

An unexpected advantage to the modularization of our algorithm was that it offered an easy way to refactor the code once it was mostly finished. Indeed, we had originally been instantiating lists of lists within multiple calls of a function, and we recognized that many of these data structures could exist as fields in the class itself, thus saving the repetitive iteration that was slowing down our code. As a result of these changes, our code's asymptotic run time was reduced from $O(n^4)$ to $O(n)$.

Django

To better integrate the back-end analysis with the web-front end, we opted to use Django - a high-level Python web framework which utilizes the Model-View-Template architectural pattern. Django aims to reduce the work needed to create complex, database driven websites using a modular component structure and a don't repeat yourself (DRY) approach. Django's clean separation between its database layer and application layer also allows its applications to be very scalable, a key consideration when assessing the future of our site.

Model View Template Architectural Pattern

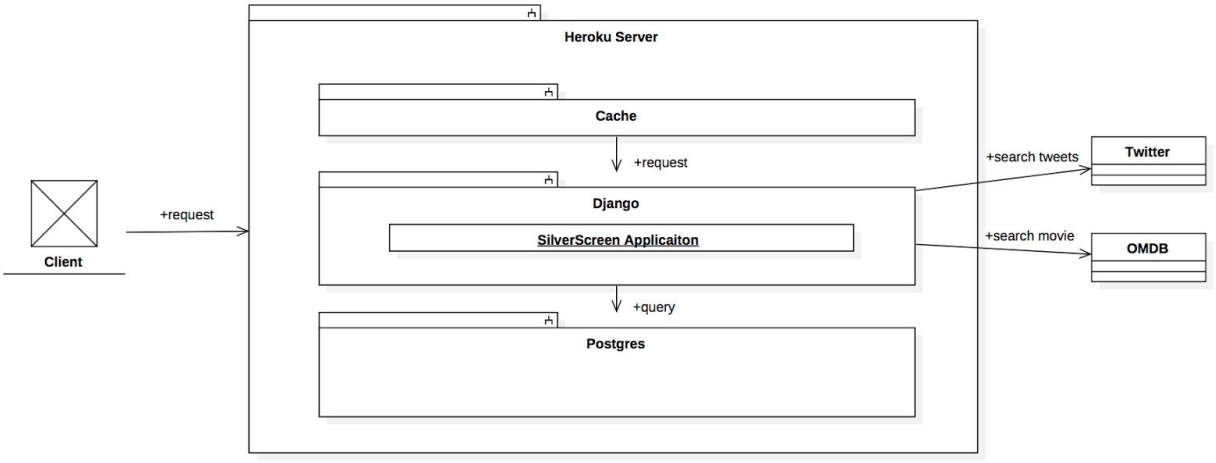
Django is based upon the Model View Template framework, a reimagining of the traditional Model View Controller framework. In this interpretation, the 'view' describes the data which is presented to the user, rather than how the data is presented to the user (a task delegated to the 'template'). In contrast with MVC, the controller (the part of the system which manipulates the model through user) is thought of as the framework itself - containing the logic to tie the model and view together.

Heroku and PostgreSQL

In order to host our application, we opted to utilize Heroku - a cloud platform which provides a complete set of tools to build and host web applications in a fully managed runtime environment. In addition to its robustness and scalability, a major factor in our decision to use Heroku was the platform's integration with a number of development tools, including Git/GitHub.

As a result of our decision to use Heroku, we were required to use PostgreSQL - a relational database management service which supports a large majority of SQL constructs. Very similar to MySQL, PostgreSQL excels in reliability and stability and requires no licensing fees to operate.

Dynamic View and Description



System Figure 1: Dynamic System View

The diagram above provides a high level overview of how the main modules of the system interact with each other after a user makes a request.

Data Management

As Silver Screen is extremely statistics-focused, the importance of data structures and their management is difficult to understate. Data was collected from three sources: Twitter, IMDb, and Rotten Tomatoes. Data needed for handling requests from the user was stored in a PostgreSQL database.

Rather than the developer creating and maintaining database tables directly, Django's database abstraction with the `psychopg2` package allowed us to access and store information through object-like "models". For Silver Screen, we opted to create three different types of models, one representing a single `Movie`, one representing a single `Tweet`, and one representing a single sentiment score (`Sentiment`) (further details about these models can be found in the following section). In each of these models' declarations, we declared the fields (and their corresponding datatypes) where the data was stored, much like within a traditional database. These field declarations were extracted by Django (through the use of the `makemigrations` python command) and a PostgreSQL database was constructed automatically from these migrations. As a result of this unique object-based approach to database management, upon the creation or update of a model we simply called Django's `save()` function on the object itself, rather than having to update the database through a SQL statement. This structure also allows us to be database independent, increasing the portability of our code.

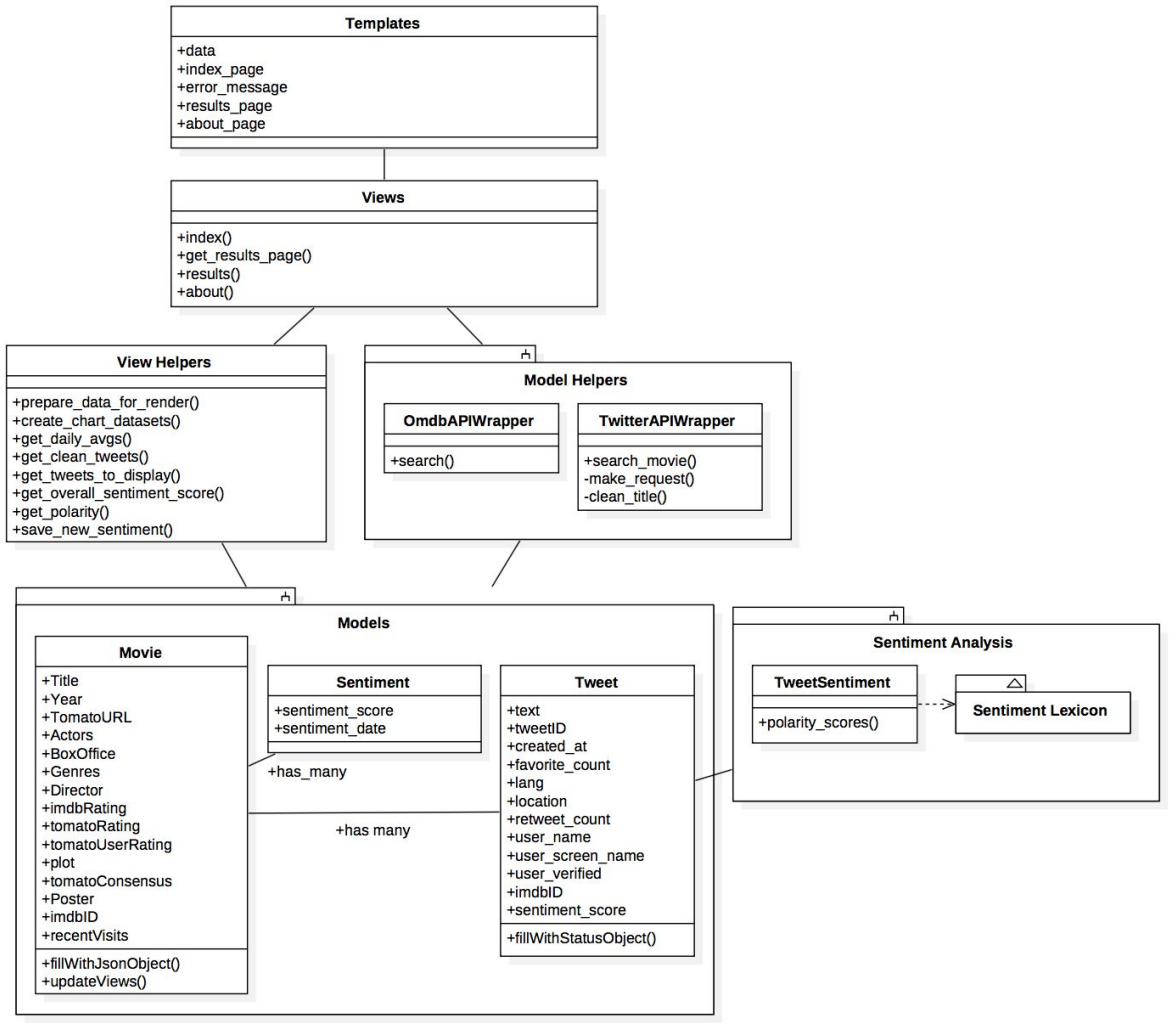
	created_at	favorite_count	lang	retweet_count	source	text	user.name	user.screen_name	user.time_zone	user.verified
1	Fri Oct 14 10:58:50 +0000 2016	75	en	40	<a href="http://twitter.com" rel="nofollow">Twitter W...	Conversation with @Potus about Star Trek, The Marti...	Joi Ito	joi	Eastern Time (US & Canada)	TRUE
2	Fri Oct 14 15:01:07 +0000 2016	68	en	38	<a href="http://www.socialflow.com" rel="nofollow">...	Matt Damon calls 'Martian' director Ridley Scott the "Ji...	Hollywood Reporter	THR	Pacific Time (US & Canada)	TRUE
3	Fri Oct 14 15:25:53 +0000 2016	42	en	24	<a href="http://twitter.com" rel="nofollow">Twitter W...	Ridley Scott will receive the American Cinemathequ...	Rebecca Ford	Beccamford	Pacific Time (US & Canada)	TRUE
4	Sun Oct 16 03:22:46 +0000 2016	NA	en	NA	<a href="http://instagram.com" rel="nofollow">Instag...	Digital Martian World Travel x Opulent Lifestyle - Thai...	A Wills	FortunePowers	Pacific Time (US & Canada)	NA

System Figure 2: Tweets pulled from Twitter using the search term 'The Martian', shown after removing extraneous columns

We originally were aiming to use NumPy and Pandas dataframes to analyze our data, but found that Django's object-style data encapsulation provided more than enough analytical power for our purposes. Using the complex database queries facilitated in addition to relational constructs provided by PostgreSQL (such as many-to-many relations, which can relate movies to one another), we were able to perform rich analysis on our data while reducing the amount of clutter and overhead in our code.

Detailed System Design

Silver Screen's architecture can be broken down into four distinct 'subsystems': models, views, and templates, in addition to our sentiment analysis modules.



System Figure 3: High-Level UML Diagram

To increase the accessibility of low-level sentiment analysis details to the general public (who might not necessarily read the entirety of this document), we have decided to focus the discussion on low-level sentiment analysis within the project's wiki.

Models

As described in the *Model-View-Template* section, the Model class (contained within `models.py` and its corresponding `helpers.py`) aids in storing the persistent state of data within Silver Screen. Each class within `models.py` represents a distinct 'object' within the realm of our sentiment analysis process - a Movie, a Tweet, or a Sentiment Score.

Movie Object

Each movie object (`Movie`) represents a distinct movie which has been historically searched for on Silver Screen. `Movie` generally mirrors the fields we retrieved from OMDb, mainly the `title`, `imdbID` (a unique identifier), IMDB/Rotten Tomatoes rating, and so on. Additionally, any given `Movie` object contains a field for recent views, an integer field detailing the number of times a movie has been accessed since midnight, updated using the `updateViews()` method (a Heroku background job clears this field nightly).

Upon initial construction, a `Movie` object contains no data - data must be passed in through a JSON object, returned by the `OMDbAPI` wrapper, containing a movie's relevant fields. The `fillWithStatusObject()` method within the Movie class completes this task, cleaning/converting data when necessary (as data within the JSON objects are strings).

Tweet Object

Similar to how a `Movie` object represents distinct movies from OMDb, a `Tweet` object represents a distinct Tweet which we pulled through the Twitter API. The `Tweet` object is also constructed in a similar manner, populating its fields through its corresponding `fillWithStatusObject()` method. In addition to the tweet's corresponding `text`, `tweetID`, and `language`, the `Tweet` object stores a list of movies which it is related to (represented by a many-to-many Django database relationship). It is important to note that an individual tweet is never represented by more than one `Tweet` object in order to save space within our database - if we attempt to construct a tweet object which is already in the database, we simply return the existing `Tweet`.

Sentiment Object

Rather than storing a movie's sentiment within its corresponding Movie model, sentiment tracking is delegated to a dedicated `Sentiment` object (allowing for historical storage of sentiment scores). Upon the generation of a new set of sentiment scores for a movie, a corresponding `Sentiment` object tagged with the movie's `imdbID` is created and stored within the database. Upon a movie's lookup on the front-end, the most recent sentiment score corresponding to that movie is retrieved (if it exists and is recent) and displayed. This sentiment

storage methodology reduces the amount of processing required in cases of repeated lookups - allowing us to reduce the amount of time spent reprocessing sentiment scores.

Model Helpers

Model helpers are the API wrappers our application uses to abstract away communications with its external APIs: Twitter and the Open Movie Database.

TwitterAPI

The `TwitterAPI` helper is a class solely responsible for communication with Twitter in order to retrieve relevant tweets about a given movie. Since calls for sending and retrieving data over the web using Twitter's API are blocking calls it is important that we streamline this process as much as possible. Therefore, our `TwitterAPI` makes multiple asynchronous requests to Twitter in parallel, so the total wait-time for multiple requests to Twitter is only as long as the slowest request.

OMDbAPI

In order to reliably retrieve a movie from OMDb we must issue a GET request with the exact name of the movie, otherwise the request will fail. Given that the user will be inputting the name of the movie there must be a way of ensuring the title will be recognised by OMDb when the request is made. Our application has two ways of dealing with this. The first is to check whether user's query closely matches any of the titles stored in a list of titles local to the `OMDbAPI` class using Python's `difflib` package. If a close enough match is found we simply issue the GET request with the matched title. Otherwise we offload this responsibility to OMDb using `search_movie()`. This returns a list of movie titles from which the most relevant match is chosen to be used in the GET request to OMDb. Of course, this is much slower than the `difflib` method (because it requires one extra request to OMDb) but serves its purpose as a backup on the rare occasions that the first method fails.

Views

The views module encompasses much of the control flow of a standard visit to Silver Screen. Driven by requests from the user, the views class retrieves appropriate models depending on what is needed to fulfill the user's request. `Views` contains one method per webpage, in addition to a handful of helper functions used across all pages. For instance, in the case where a user requests the sentiment analysis of a given movie (represented by the `results()` method), the views class utilizes the OMDb/Twitter APIs, local database, and appropriate helper functions to collect the given movie's basic information and sentiment results and forward it to the template (an exact control flow is detailed below).

As a side effect of handling user requests, the views class is also responsible for user-level error handling. In the event where an unhandled exception occurs in the retrieval of a movie's data,

the exception is caught in the views layer, the exception is logged in console, and the user is moved to a error page containing a high-level, user-friendly description of the error.

Control-Flow Overview

Data flows in the system is as follows, with each indentation representing a new layer:

- Client submits query
    - Query submission calls `results` method
        - If a random movie is requested, the `imdb-pie` module retrieves a list of the top 250 movies on IMDB, and we send a request using a random title from those returned
        - Otherwise `results` method calls `OMDbAPI` Wrapper
            - `OMDbAPI` Wrapper is driven using the `omdb` library, retrieving data from The Open Movie Database
                - ➢ IMDB and Rotten Tomatoes information is returned in JSON format
            - `OMDbAPI` Wrapper parses JSON response and stores the most popular movie retrieved as a Movie object
        - `results` calls `TwitterAPI` Wrapper
            - `TwitterAPI` wrapper makes Twitter API request
                - ➢ Twitter returns JSON response
            - `TwitterAPI` wrapper parses JSON response and returns a list of Tweet objects, ensuring that each tweet is tagged with the corresponding `imdbID`
                - ➢ `Tweet` constructor calls `TweetSentiment` to assign a sentiment score to each tweet before storing it
        - `Results` calls a series of `helpers` to filter and reduce the data, and return it to the client
    - Data is displayed on the client side

Templates

To interface with the end-user's web browser, Django's template system dynamically modifies specially constructed HTML files in order to display custom content. While Django's template constructs allow for control statements (i.e. *if()*) and loop constructs (i.e. *for()*), it is important to note that these statements are only used for populating the template (they are evaluated server-side and a static HTML file is delivered to the user). This restriction requires the use of additional JavaScript to control dynamic portions of the site (i.e. the sentiment charts).

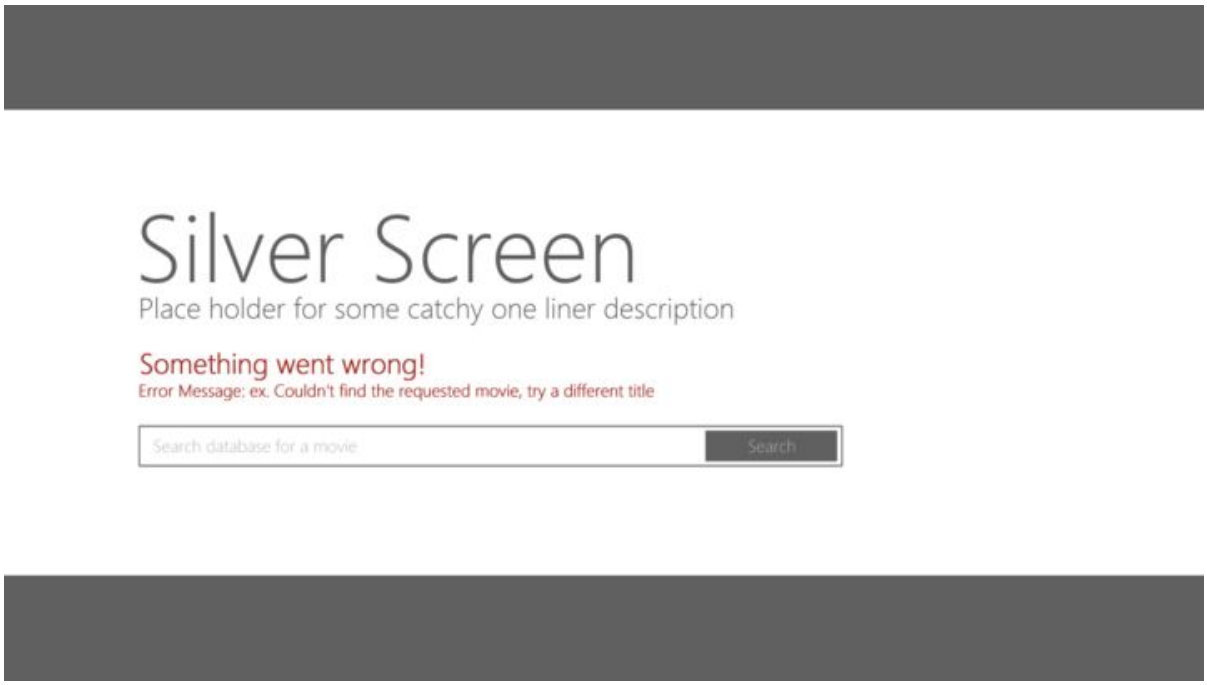
In order to maintain a cross-platform, user-friendly UI, we opted to utilize Bootstrap to design a majority of our site. In addition to delivering a modern user interface, Bootstrap allowed us to offload a large portion of CSS development to its libraries, reducing web development time as a result. Bootstrap's tooltip functionality also allowed us to hide non-essential information from the user, reducing clutter and helping maintain a clear user interface.

We considered a multitude of options when determining which charting platform to use when displaying sentiment analysis breakdowns. We ultimately decided on Chart.js, an open-source JavaScript-based charting platform, due to its pertinent chart types, animation capabilities, and all-around look and feel. Much like Bootstrap, Chart.js also allows for integrated tooltips, allowing us to incorporate detailed data (such as a tweet's full text) which would otherwise be excluded for brevity.

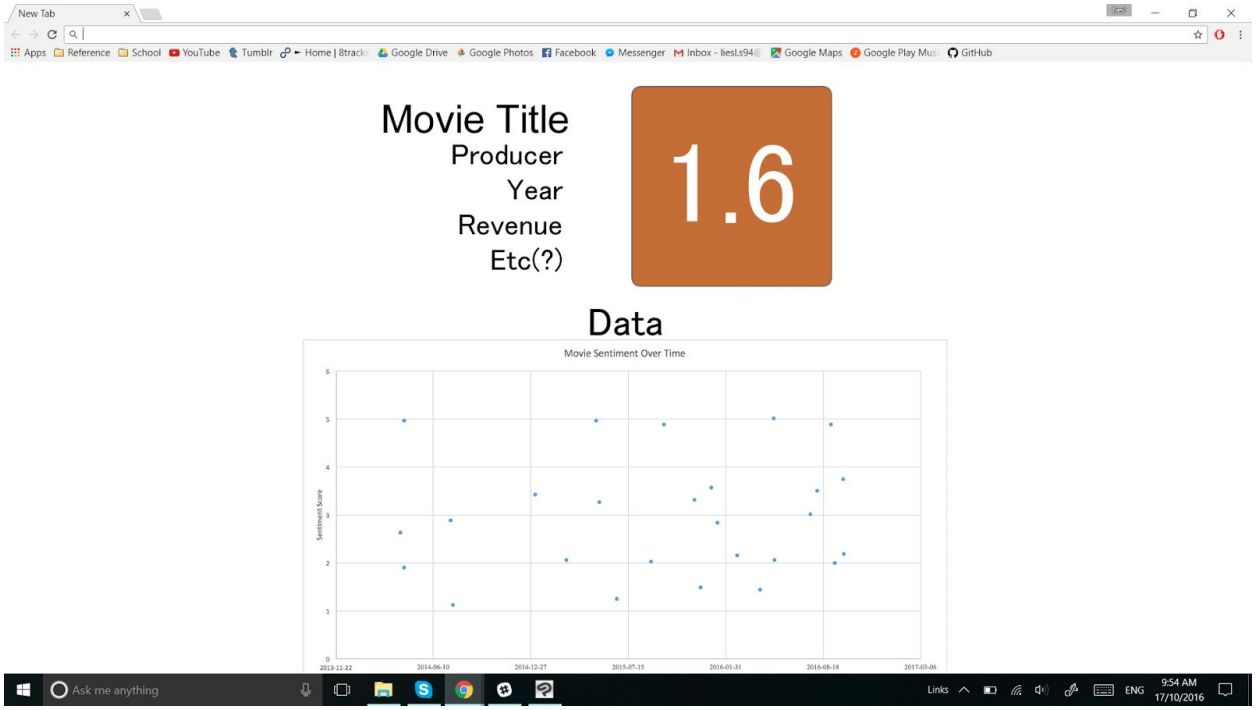
GUI

Initial Mockups

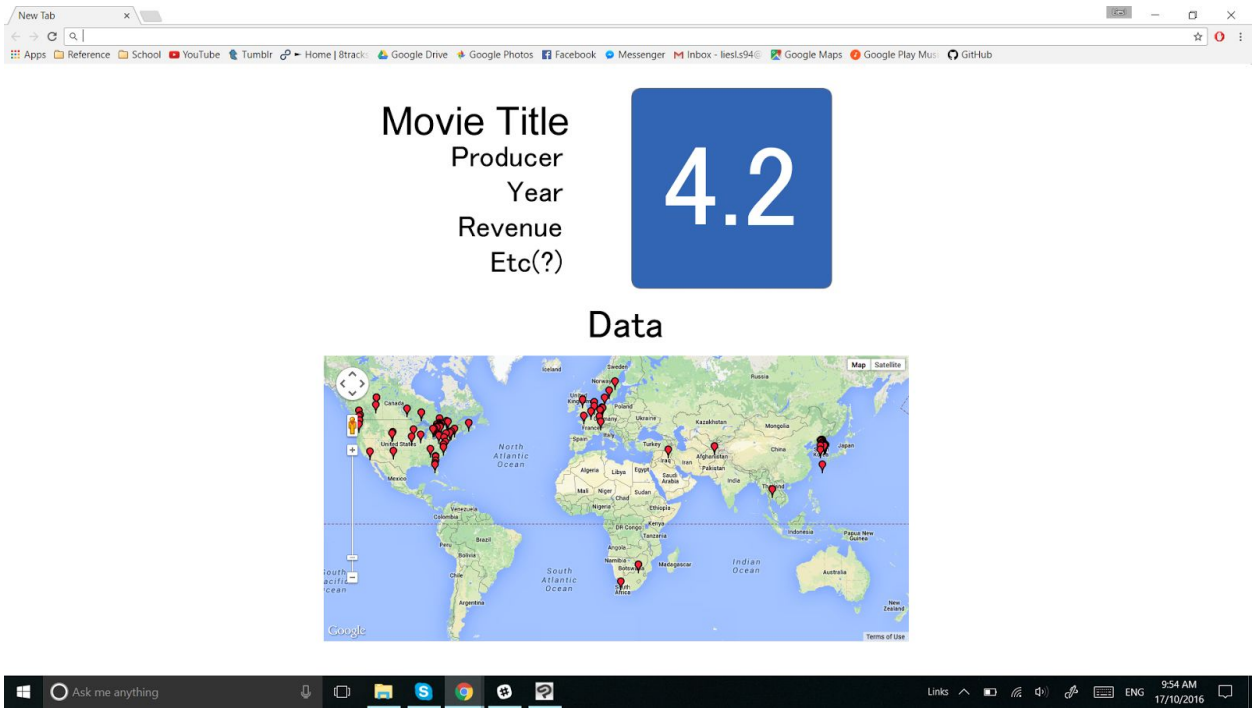
UI Figure 1: Index Page Mockup



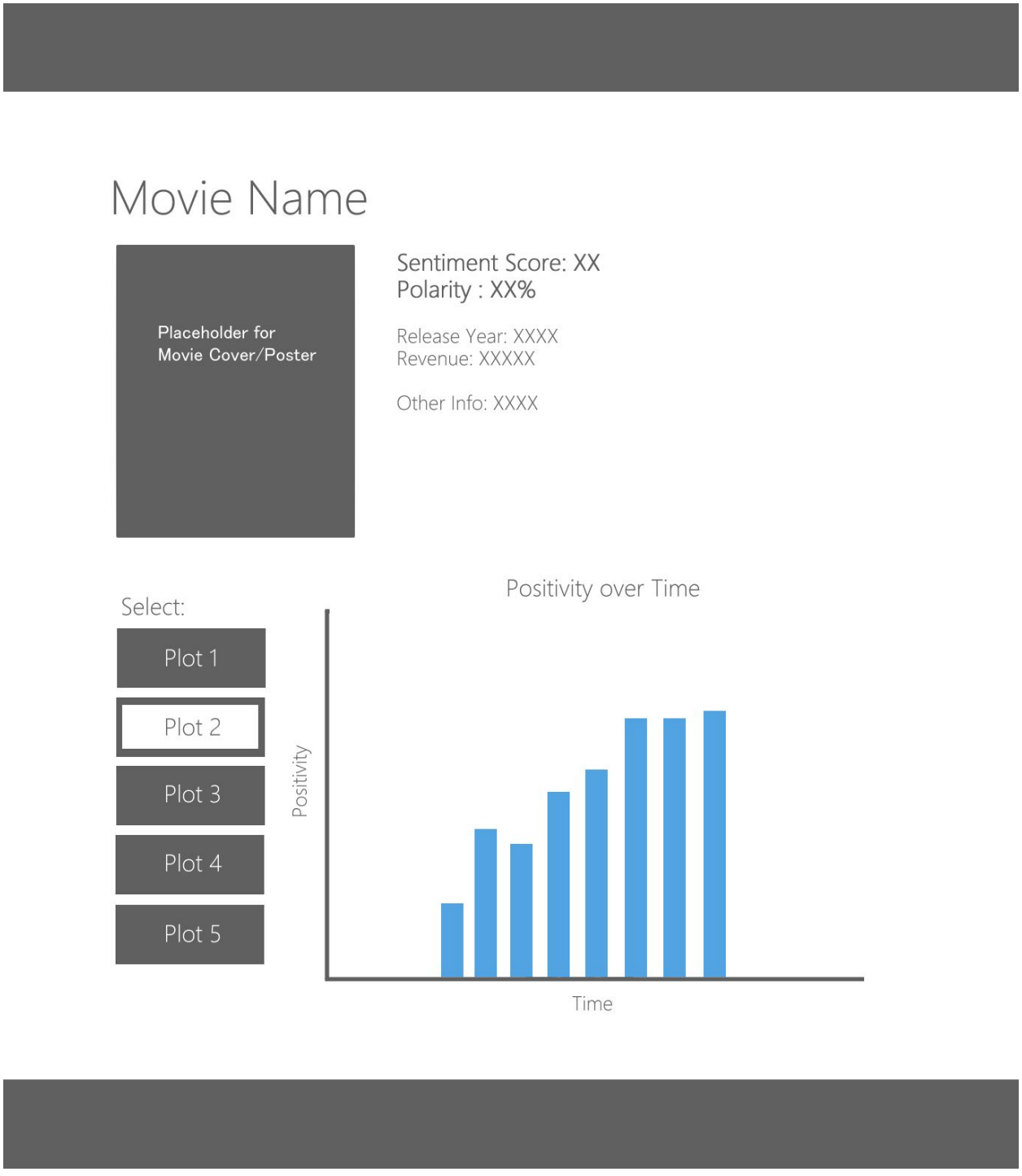
UI Figure 2: Error Case Mockup



UI Figure 3: Sentiment Score Presentation Mockup



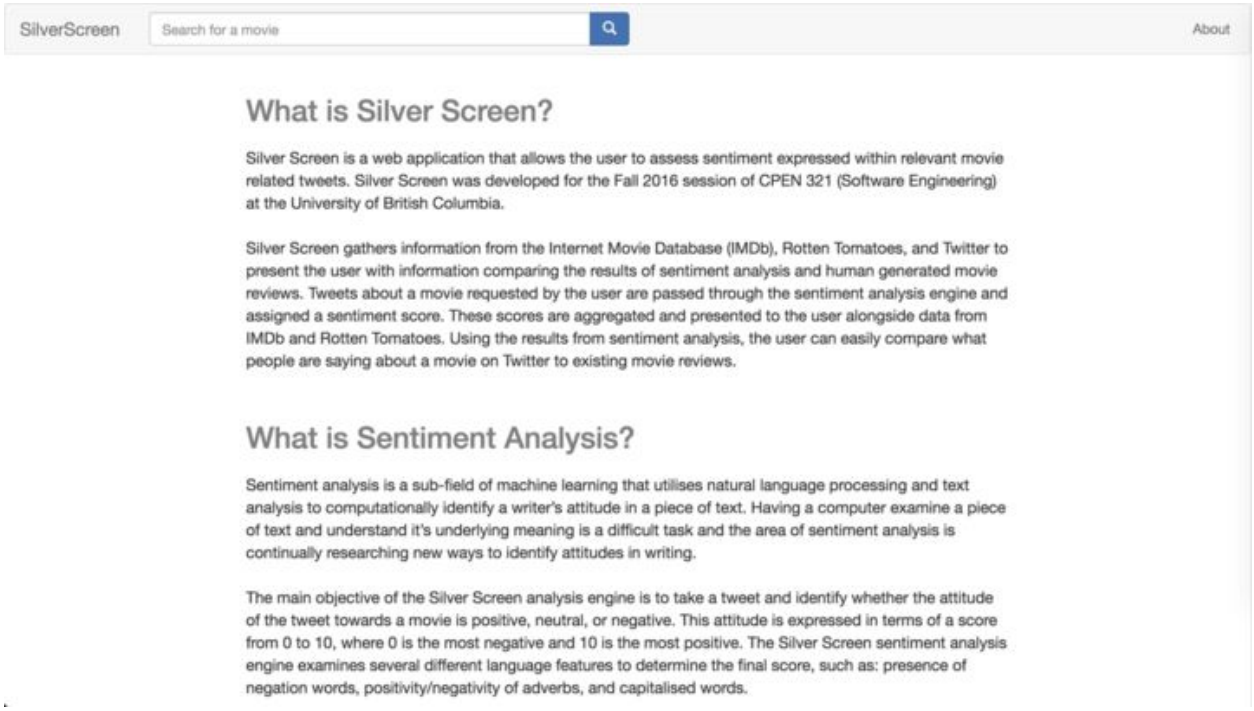
UI Figure 4: Sentiment Score Presentation Mockup
(with unpursued geographical mapping feature)



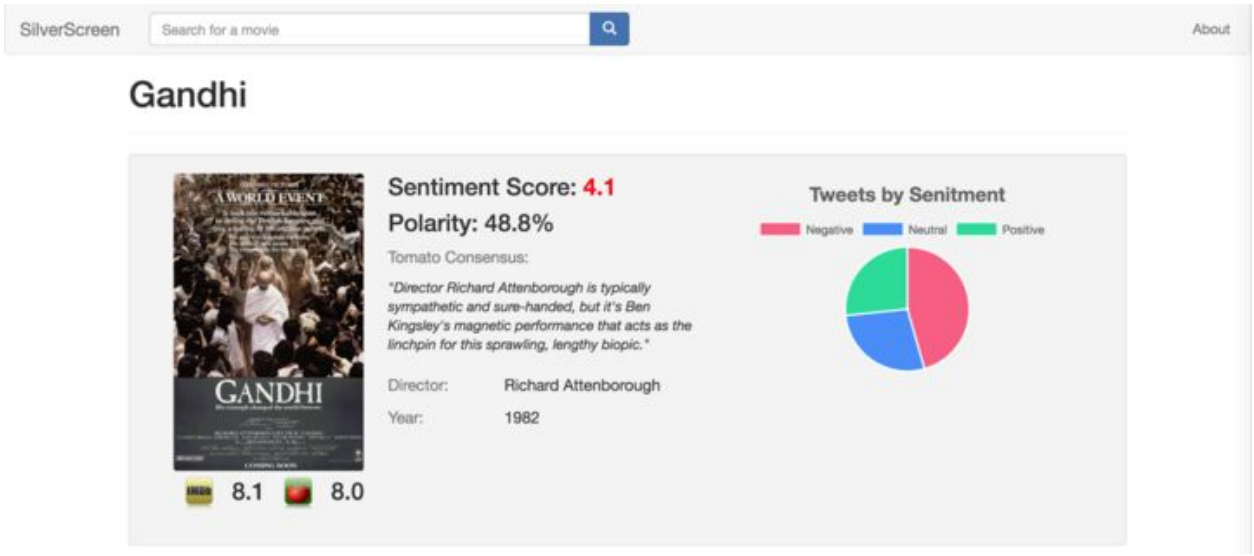
UI Figure 5: A later mockup for the results page which incorporates tabs to display multiple graphs and displays more information returned from IMDb and Rotten Tomatoes.

Final UI Design

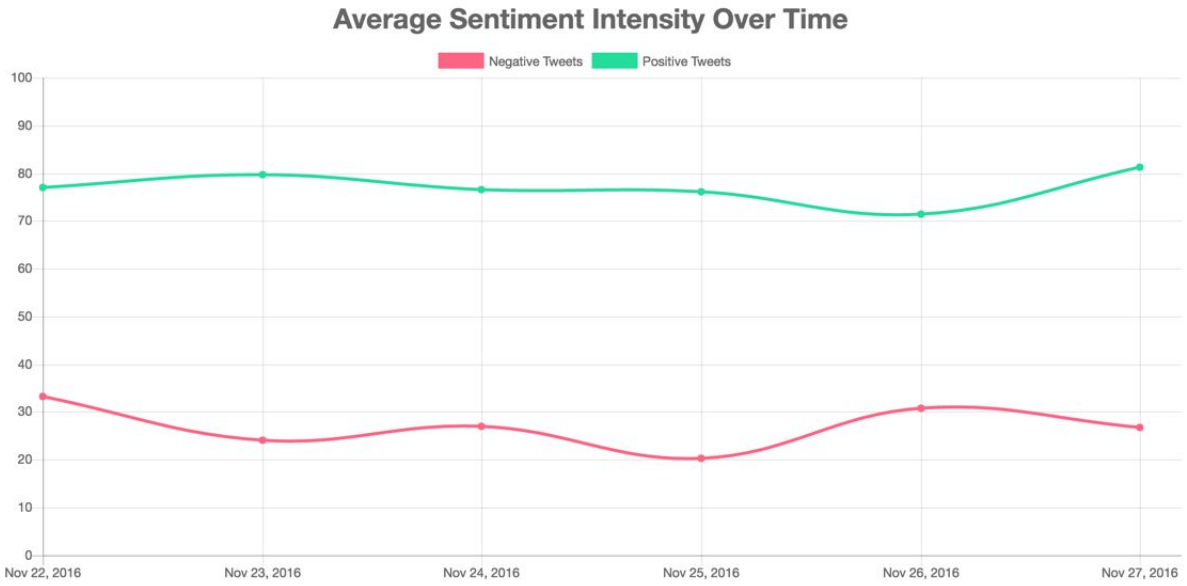
UI Figure 5: Index Design



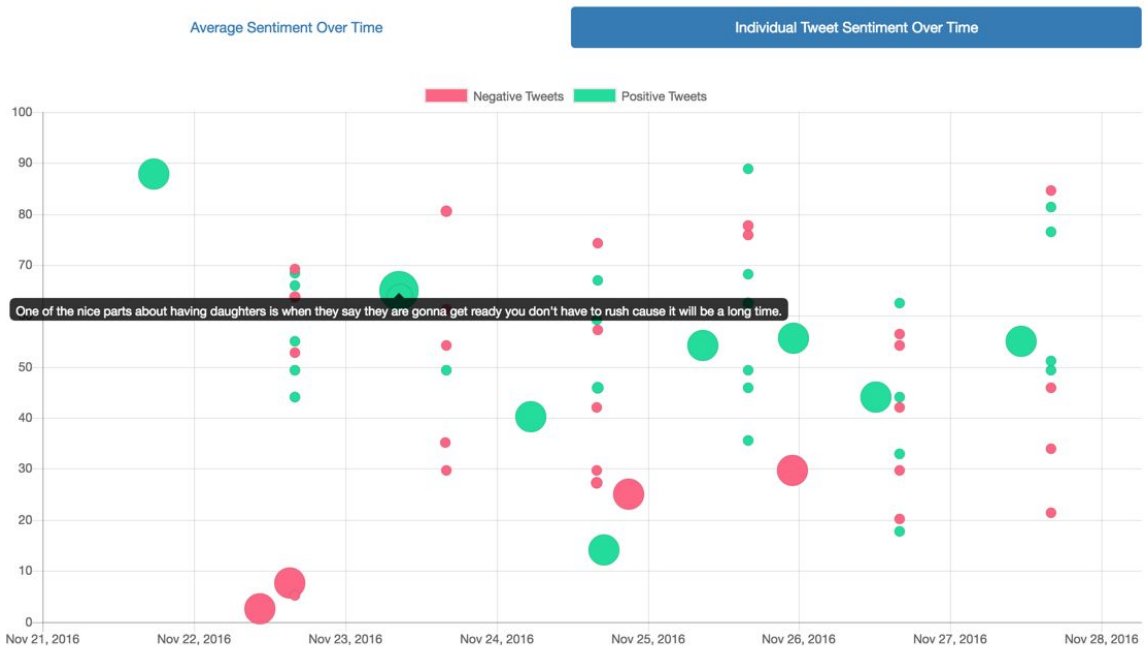
UI Figure 6: About Page Design



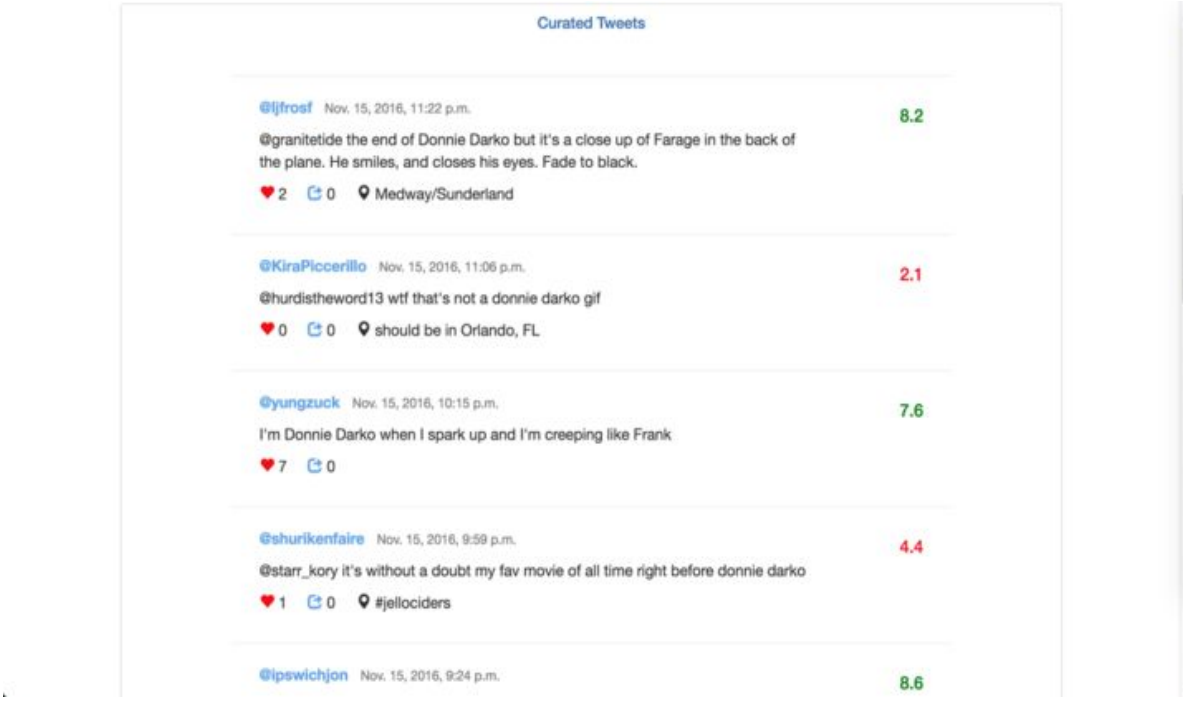
UI Figure 7: Sentiment Score Presentation



UI Figure 8: Average Sentiment Intensity Graph



UI Figure 8: Individual Tweet Sentiment Graph with Tweet Display on Hover



UI Figure 9: Tweet Breakdown Graph

Our main focus in developing Silver Screen's UI was to maintain a high level of usability for users of all skill levels, while still providing a detailed set of data for user inspection. In order to accomplish this, we opted to create a fairly minimalistic website on the surface, with a large amount of information hidden behind drawers, charts, glyphicons, and other Bootstrap constructs.

Validation

As Silver Screen is aimed towards a large segment of the population, we were very cautious in assuming the skillsets and expectations that our users might have had. Resultantly, it was very important that we developed our application with continuous feedback from outside sources - especially from those outside of our own technical background. We considered reaching out to the alternative users described in our requirements document (mainly, industry professionals and theatre operators), however due to time constraints we opted only to consult standard users.

We decided that it would be best to make a survey that we could distribute to our friends and family to collect user opinions. Our survey consisted of 8 questions to get users to think about our UI design and website functionality. Overall we received 12 detailed responses that helped us develop our about page, improve how we display data on our graphs and modify the naming of parameters. Our survey can be viewed here, and its results can be found here.

Design Changes and Rationale

As development of Silver Screen progressed, there were been multiple key developments which warranted the modification of this document. They are listed below (key changes are italicized).

Removal of NumPy and Pandas Discussion
We have opted to forgo the use of the NumPy and Pandas statistical analysis toolsets which were originally described in this document in favour of use of Django's built in model tools. While NumPy and Pandas are both very powerful, we have found in development that attempting to port information from the Django/Postgres dataframe to NumPy/Pandas is inefficient for our use and ultimately unnecessary. While these packages are more powerful in nature, Django's native model toolkits (allowing us to filter/manipulate data) are more than enough for our needs.

Modification of NLTK's Role
While NLTK was originally planned to be an integral portion of Silver Screen, it's use has been significantly curtailed in favour of our own custom-made sentiment analysis modules (which we have found to be more accurate when analyzing tweets). Details on the transition (and further rationale) can be found in the Sentiment Analysis portion of this document.

Clarification of API Wrappers
The original version of this document briefly mentioned the use of API wrappers to retrieve IMDB and Rotten Tomatoes data. The document has since been updated to provide more insight to this portion of our project.

Detailed MVT Discussion
In addition to the pre-existing discussion of Django's MVT structure, an additional section detailing Silver Screen's low-level implementation details (including the overview of internal classes and methods) broken down by their position within Model/View/Controller framework has been added.

Expansion on Database Structuring Discussion
As features have continued to be implemented, our database structure has been significantly updated since the publishing of the preliminary design document. An updated overview of the database, including rationale, can now be found in the Data Management section of this document.

Additional Sentiment Analysis Discussion
Please note that further low-level documentation regarding the sentiment analysis module has been placed within Silver Screen's GitHub wiki for better accessibility, and can be found here: https://github.com/bfbachmann/Silver-Screen/wiki/Sentiment-Analysis-Guide