Electronics and Computer Science
Faculty of Physical Sciences and Engineering
University of Southampton

Jamie Davies
14th May 2014

A Non-Personalised Approach to Twitter Hashtag Recommendations

A project report submitted for the award of
MEng Computer Science

Supervisor:
Dr. Nick Gibbins

Examiner:
Dr. Klaus-Peter Zauner

Abstract

One of the key characteristics of Twitter and other microblogging platforms is the use of 'hashtags' — topical/categorical annotations provided by the authors of the posts (tweets) themselves. This flexible system was designed for the effective organisation and searching of tweets, but with Twitter facing an ever-increasing number of users and tweets it is hard for users to keep track of the vast number of hashtags in popular use. This results in data from the hashtags being fragmented and inaccurate due to the users making poor or uninformed hashtag choices.

This project presents an intelligent hashtag recommendation tool for use with Twitter, that makes it easier for users to compose tweets with relevant hashtags, as well as aiding the search and navigation of Twitter as a whole. The presented technique integrates a Naive Bayes classifier with a novel stream processing framework to give functional, non-personalised hashtag suggestions and search query expansions.

Contents

Acknowledgements

Thank you to everyone involved. Most importantly, thank you to Nick Gibbins for providing the inspiration for this project. Thank you to Jon and Sina for giving up their time to give me the advice and enthusiasm I needed. Thanks to Katie for having the patience to be optimistic when I couldn't remember how. Thank you to Hayden and Miles, for making the LaTeX styling of this document a constant competition. Thanks to Mum and Dad for their support and proofreading skills. Finally, thanks to everyone who downloaded and used Spout, the response to the package was overwhelming.

Statement of Originality

The work contained within this report has not been previously submitted for a degree or diploma at any other higher education institution. To the best of my knowledge and belief, this project contains no material previously published or written by another person except where due references are made.

Chapter 1

Introduction

Collaborative tagging (sometimes referred to as social tagging) systems are designed to allow users to navigate, organise and manage online resources. The central idea behind these systems is the concept of an annotation: the action of a user applying a personalised tag to a resource. On a large scale, many of these annotations can be taken together to form a complex network of users, resources and tags — commonly referred to as a 'folksonomy' (Xu et al., 2008).

One of the most popular applications of a folksonomy is Twitter[1]. 'Hashtags', which are simply any combination of letters or digits preceded by a hash character (#), afford users the ability to create free-form tags to describe the content of their posts ('tweets'). Tweets can be categorized with several hashtags, thereby creating networks of tweets and users, and making it easy to find other related tweets, users and hashtags. This renders hashtags as a powerful tool in aiding the search and navigation of the billions of tweets contained within Twitter.

However, despite its numerous benefits, the hashtag system presents new challenges to overcome before it can become truly useful. Due to the open nature of folksonomies, it is important that users have the freedom to create and use exactly the tags they wish to use. However, this unsupervised tagging can result in vast numbers of hashtags for users to choose from, often including redundant or ambiguous hashtags. When posting a tweet, there is nothing stopping a user from creating an entirely new hashtag to describe something with exactly the same meaning as a collection of other hashtags. This redundancy can confuse users and fragment the true meaning behind the synonymous hashtags.

Evidence of this problem can be easily found through a casual navigation of popular hashtags or topics on Twitter. It is commonplace for users to use a collection of tags in one tweet that are all synonymous. An example of this can be seen from an arbitrarily chosen tweet (with the author's personal details removed) in Figure 1.1.

---

[1] `www.twitter.com`

Figure 1.1: An example showing the tagging practices used on Twitter.

1.1 Project Goals

The goal of this project is to develop an intelligent hashtag recommendation system for use with Twitter. The objective is to make it easier for users to select relevant hashtags for their tweets, and in doing so, begin to tackle the problems presented by the sheer quantity and redundancy of hashtags in Twitter. The aim is that the user will be able to see hashtag suggestions in real-time as they are typing a tweet, thus increasing the likelihood of them publishing tweets with accurate hashtag data.

Such a recommendation system would also need to aid users in the navigational and exploratory aspects of folksonomies. Therefore, another objective for this project is to make it easier for users to navigate and search the fragmented hashtag data already published on Twitter. The aim is that the same recommendation system used to suggest hashtags for content a user is creating could also be applied to create enhanced search queries to retrieve tweets from Twitter.

Chapter 2

Background Literature & Related Work

The motivation behind hashtags is to categorise tweets and allow them to show up more easily in searches[1]. Whilst the task that this project is aiming to complete is novel and fairly unexplored, it is well connected with other experiments, systems and projects within the research community.

2.1 Traditional Recommendation Systems

Traditional recommendation systems are in place all over the web today. From music discovery services (such as Last.fm[2], shown in Figure 2.1(a)) to suggested purchases on retail sites (like on the front page of Amazon[3], shown in Figure 2.1(b)), these systems are all personalised recommendation engines that take an individual user's preferences and use them to provide suggestions tailored to that user.

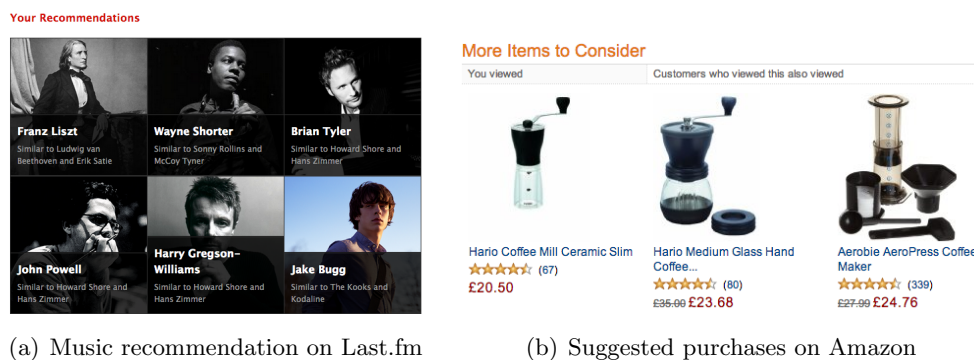(a) Music recommendation on Last.fm   (b) Suggested purchases on Amazon

Figure 2.1: Examples of traditional recommendation systems.

---

[1] https://support.twitter.com/articles/49309-using-hashtags-on-twitter
[2] www.last.fm
[3] www.amazon.co.uk

Most personalised recommendation systems employ a set of techniques known as collaborative filtering. These techniques were first coined by Goldberg et al. (1992), where a system named *Tapestry* was created that allowed people to attach annotations to documents, and then use that information to filter the documents for other users.

One common implementation of collaborative filtering is the so-called "user-to-user" approach. "User-to-user" collaborative filtering works by taking the preferences of a user A and finding a small subset of other users in the system that have similar preferences. For each user B in the subset, any items that B has adopted that A has not are added to a ranked list of suggestions. A is now more likely to adopt items in the list than the items of another random person (Schafer et al., 2001).

Another approach to provide relevant recommendations to a user is the use of content-based recommendation systems. This is a type of system that recommends items relevant to other items by comparing their details and descriptions. This can be extended to suggest items for a user by comparing a content-based description of the user's preferences with the descriptions of the items (Pazzani and Billsus, 2007).

A key issue with content-based filtering is that the recommendations can only be as accurate as the algorithm used to derive a user's profile. There are a number of algorithms available to build user profiles, depending upon the context, but essentially a content-based profile is created using a weighted vector of item features. The weights mark the importance of each feature to the user, and can be computed from individually rated content vectors.

Cantador et al. (2010) studied and evaluated a number of content-based recommendation models based upon the premise of user and item profiles being described in terms of weighted lists and tags. Through their experiments they found that models that focused on user profiles outperformed the models oriented towards item profiles in nearly every test. They go on to suggest that a better way of profiling users would be through the use of tag clustering.

Instead of limiting recommendation systems to the accuracy of their classifiers, a common approach is to incorporate relevance feedback techniques. Relevance feedback is a process that was originally designed for information retrieval, and works on the assumption that a user cannot always correctly encapsulate into a query what it is they are searching for. It works by allowing a user to create a query to which an initial set of results is returned. From these, the user can then mark certain results as relevant or irrelevant. This information is then submitted and used to refine the original query and return more relevant results to the user (Salton and Buckley, 1990).

Utiyama and Yamamoto (2006) showed that it is possible to combine collaborative filtering, content-based filtering and relevance feedback techniques into one system to provide better recommendations.

2.2 Assigning Labels with Machine Learning Techniques

Although there exists a long history of research into traditional recommendation algorithms, the data structure of a folksonomy is distinct from those found in traditional problems. This has led to a new generation of supervised learning classification algorithms which can be applied to the problem of assigning labels to resources.

In order to facilitate the comparison of resources, tags and users, aggregate projections of the data can be constructed. This reduces the dimensionality of the data by sacrificing some information (Schmitz et al., 2006). There are two main classes of algorithms that perform this task: feature *selection*, which reduces the size of a dataset by selecting a subset of the data, and feature *extraction*, which uses the existing data to generate entirely new features. Despite the improved results that feature extraction can provide, their computational cost and memory requirements can often make them impractical for use on the extremely large datasets common with folksonomies (Gemmell et al., 2009).

One compromise to this trade-off is to use an algorithm such as Hebbian Deflation, which approximates the outputs of the intensive feature extraction techniques, but with lower computational cost and memory requirements (Oja and Karhunen, 1985). Gemmell et al. (2009) used this approach coupled with a k-Nearest Neighbour classifier to find appropriate tags for a given resource. This was done by finding 'neighbouring' users: "Neighbours are selected only if they have tagged the query resource and tags are selected for the recommendation set only if they have been applied by the neighbour to the resource" (Gemmell et al., 2009). It was tested on data from the popular bookmarking site Delicious[4] and from Citeulike[5], a tool that enables researchers to manage and discover scholarly references, and it was found to be a successful approach with giving results.

Niwa et al. (2006) took a different approach to overcome the limitations of collaborative filtering. Using a modified version of TF-IDF (Salton and Buckley, 1988), they calculate the affinity level between users and tags (where affinity level is a measure of how frequently a user uses a particular tag), and then use this to calculate the similarity between tags. These similarity measures are then clustered using a custom clustering algorithm to provide a set of recommended tags for a given user and resource. This enabled them to adjust the level at which a user's preferences were taken into account when determining recommendations, as well as providing a simple solution to the tag redundancy issue found across most folksonomies.

Another popular machine learning approach to the folksonomy tag recommendation problem is the use of a Naive Bayes classifier. Naive Bayes has been an accepted tool in information retrieval for a long time, with some early literature dating back over 50 years (Maron and Kuhns, 1960), and has recently been seeing a renaissance within the machine learning community.

As shown by De Pessemier et al. (2010), Naive Bayes can be easily adapted and extended to work in the context of providing recommendations to match a query

---

[4]http://delicious.com/
[5]http://www.citeulike.com/

or a set of user preferences. In particular, their algorithm relies upon the general user preferences for cases where the classifier produced an uncertain outcome, and becomes more context specific on a sliding scale as the classifer gives more certain probabilities.

2.3 Approaches to Hashtag Recommendation

Even though providing hashtag recommendations and suggestions is still a new and largely unexplored field, there have been several efforts to improve the hashtag experience for Twitter users.

The current hashtag system on Twitter (Figure 2.2) uses a non-personalised auto-complete tool to provide suggestions to the user. Whenever a hash symbol (#) is typed in the tweet composer, the system suggests hashtags starting with the letters that the user has typed so far. These suggestions are chosen from a tiny subset of hashtags, taken from a mixture of those currently trending[6] and from the user's history. Whilst better than not having suggestions at all, this system is only truly useful in one of two specific use cases: when the user knows the starting letters of a trending hashtag they want to use, or are trying to recall a hashtag that they have previously used. This system does not help the user choose the correct hashtag for the content of their tweet.

Figure 2.2: Twitter's current hashtag suggestion system.

By assuming that the primary purpose of hashtags is to categorise tweets and improve searching (as Twitter envisioned), Zangerle et al. (2011) created a system that recommends hashtags for a tweet by taking hashtags from other tweets that are textually similar to the query. The similarity between tweets is calculated with the TF-IDF model. The hashtags are then extracted from the similar tweets, ranked according to their similarity to the original query, and returned as a list of suggestions to the user. A number of different ranking algorithms were tested, but this

---

[6]Trending hashtags are those with the highest rise in usage within a given time period.

was found to be the most successful.

After studying the advantages of providing personalised recommendations in retail situations on a per-user basis, Kywe et al. (2012) realised that a similar approach towards hashtags could prove fruitful. Hashtag use varies from person to person, with some users using the latest trending hashtags, other users only using a specific set or type of tag, and with some users barely making use of them at all. They proposed a personalised hashtag recommendation system that considers both user preferences and the query tweet content: the system creates a ranked list of hashtags from both the most similar users and most similar tweets. This gave promising results, although it was noted that this may not be the best recommendation system for all types of tweets and hashtags.

Shepitsen et al. (2008) used a hierarchical agglomerative clustering algorithm to profile users and provide personalised recommendations in collaborative tagging systems. They found that clusters of tags can be effectively used to ascertain a user's interests, which could then be used in a traditional content-based recommendation approach. This technique worked well, particularly for dense folksonomies such as Last.fm.

Observers of social media have realised that hashtags play a dual role within the online microblogging communities, such as Twitter. On one hand, hashtags fulfil the design goals that Twitter created them to accomplish (bookmarking and improving search); on the other hand, however, they serve as a badge of community membership and connect users together. Yang et al. (2012) took this duality into account when attempting to create a hashtag recommendation system by training a support vector machine (SVM) classifier with a variety of features taken from the tweet metadata to overcome the duality and suggest relevant hashtags.

Chapter 3

Analysis

3.1   Preliminary Data Analysis

To investigate the usage of hashtags and the problems associated with them further, a dataset of 500,000 tweets was collected from the sample Twitter stream over a time period of approximately 4.5 days. The tweets collected were filtered to ensure that they were in English, and contained at least one hashtag.

Through the use of some trivial Python scripts, it is easy to find evidence of poor hashtag choices. In particular, the most overwhelming observation is the quantity of redundant hashtags in use throughout Twitter. Table 3.1 shows several hashtags taken from the dataset and their number of occurrences. Despite the tags all having exactly the same semantic meaning, the use of the tags is fragmented.

Hashtag	Number of Occurrences
#peopleschoice	94849
#peopieschoice	2043
#peoplechoice	439
#peoplesch	287
#peopleschoi	269
#peoplesc	230
#peoplescho	219
#peolpeschoice	164
#peopleschioce	137
#pca	94

Table 3.1: Occurrences of hashtags referring to the People's Choice Awards

Another interesting observation is of the times that different hashtags are used, which varies greatly from hashtag to hashtag. Some tags are used in tweets at a fairly uniform rate (Fig. 3.1(a)), whilst others feature large spikes of usage over a short period of time (Fig. 3.1(b)). This demonstrates the communities behind the hashtags — rather than a hashtag being used at a particular time for a particular event, some hashtags are used consistently by users to join in on a large-scale conversation.

(a) #android

(b) #blackfriday

Figure 3.1: Usage of hashtags over the period that the dataset was collected.

3.2   Requirements

As demonstrated above, there is a need for a hashtag recommendation system to improve the quality and clarity of hashtags in Twitter, as well as to aid navigation of the fragmented data that already exists. This leads to the two main requirements the system in this project must fulfil:

- Users should be able to compose and publish tweets whilst the system suggests hashtags relevant to the content of their tweets.

- Users should be able to search for a hashtag and view related tweets, including those that don't contain that hashtag.

As seen from the research and discussion presented in Chapter 2, most existing approaches to providing recommendations within a folksonomy make extensive use of user preferences in their algorithms. From calculating a 'neighbourhood' of similar users to falling back on a set of users' preferences when the data is unclear, most current approaches provide a personalised recommendation system.

This raises the question of whether hashtags are clearly defined, or whether their meanings (and therefore the tweets they apply to) are dependent upon user opinion. Through the preliminary examination of a sample Twitter dataset conducted in Section 3.1, this project is inclined to believe that the correct usage of a hashtag is independent of how frequently a particular user uses that hashtag. Indeed, it

could be argued that a user's preference towards a particular hashtag can result in incorrect hashtags being used — which is the very problem this project is aiming to tackle.

This indicates another requirement of the recommendation system:

- The system must provide identical recommendations for all users, user preferences should not be taken into account.

As with all folksonomies, there is not a predefined set of hashtags available to users on Twitter, meaning new hashtags are created and used every day. With this in mind it becomes clear that a lot of data will be needed to train the recommendation system, so that it successfully recognises and recommends a sufficient number of hashtags. This gives another requirement of the system:

- The system must be capable of training on a sufficiently large dataset of tweets, with potential to use the live data from the Twitter stream.

Lastly, Twitter is available through its web platform on practically every internet-connected device that exists today. For this system to be successfully implemented and used by Twitter users, it too must be available on all of Twitters supported devices. In addition to this, it must be as easy to post a tweet using the recommendation system as it is on Twitter.

- The system must be accessible through a web interface that provides quick and easy access to the recommendation system.

From these broad requirements, a complete set of formal requirements can be inferred:

3.2.1 Functional Requirements

1. The system must allow the user to log in and publish tweets using their Twitter account.

2. The system must provide hashtag recommendations as the user is writing a tweet.

3. The system must perform a hashtag search through a large dataset of tweets and return all relevant tweets, including those that do not contain the search query.

4. The system must use information from a large dataset of tweets to generate a model representing each hashtag.

5. The system must only use the data available through the Twitter API, and not make use of any other externally collected metrics, such as user preferences.

6. The system must be able to compare tweets against its representational hashtag models.

7. *Optional:* In place of a large dataset, the system could be able to use information from the live Twitter stream.

8. *Optional:* The system could provide probabilities for how likely a hashtag is to be related to a tweet.

3.2.2 Non-Functional Requirements

1. The system must be accessible via a web interface.

2. The system must be responsive and easy to use.

3. The system must be able to perform searches quickly.

4. The system must be able to make hashtag recommendations quickly.

5. *Optional:* The system could be able to produce visualisations to provide an easy way to interpret the hashtag recommendations/assignments.

6. *Optional:* The system could be accessible via mobile web browsers.

Chapter 4

Design

4.1 Architecture

The primary aim of the system is to produce an ordered list of recommended hashtags for a given string with a maximum length of 140 characters (a tweet). The secondary aim is to facilitate query expansion when searching Twitter for a hashtag. Due to Non-Functional Requirement 1, both of these sub-systems must be available through a web interface. It is unreasonable and infeasible to expect all necessary training and classification to occur in code running within the user's browser, especially considering that some of those browsers might be on low-powered mobile platforms. For these reasons, a client-server architecture has been chosen for the system.

Using a client-server software model will allow for a distinct separation between the hashtag recommendation system and any interfaces that make use of the tool. The server (Figure 4.1(a)) will be running a single instance of the recommendation engine, and will handle all training and classification events. Multiple clients (Figure 4.1(b)) can then query the classifier running on the server for either hashtag suggestions or a search query expansion.

Server	Client
- Hashtag recommendation	- Twitter authentication
- Search query expansion	- Twitter API calls
- Training classifier	- Send requests to server
- Persistent state	- Receive data from server

Figure 4.1: Functional overviews of the server and client.

This design ensures that there is a very light load on the clients using the tool, which is an ideal solution to enabling usage of the system from all internet-connected devices (Non-Functional Requirements 1 and 6).

Another benefit of using a centralised server to encapsulate the hashtag recommendation system is the clear separation of functionality between the client and server.

This means that the interface which the user sees and interacts with is distinct from the technical functionality — enabling multiple differently styled interfaces to connect to the recommendation server. It even presents the possibility of one day integrating with the Twitter web platform itself.

4.2   Abstract Implementation

When using a client-server execution model, it is important to correctly balance the tasks that need to be executed between the clients, the server and the users. Placing a high computational load on the users' machines can limit the accessibility of the system, whilst placing a high computational load on the server can cause a slow-down or even interrupted service for all clients using the system.
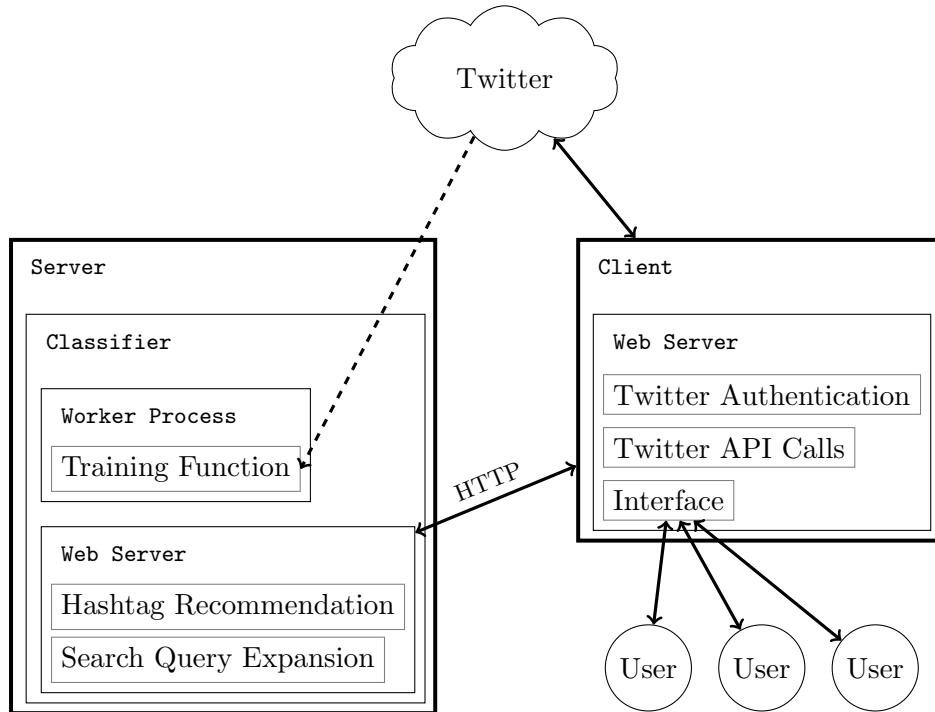
Figure 4.2: An abstract view of the implementation of the system.

The server encapsulates the hashtag classifier system. It will have the ability to train the classifier on tweet data independently and asynchronously to the other tasks that the classifier may need to perform. The training will perform as many of the computationally intensive operations on the classifier model as it feasibly can, to ensure that any requests from a client to classify a tweet or expand a query are performed as quickly as possible (fulfilling Non-Functional Requirements 3 and 4). Asynchronous training will also provide the opportunity to integrate training data from the live Twitter stream (as described in Functional Requirement 7).

The client's primary focus will be delivering the interface to the users. It will provide a way for the user to interact with the recommendations and search tools available

on the server, as well as handling the users' authentication with Twitter and calling appropriate tweet or search functions in the Twitter API.

4.3 Classification Algorithm

To choose an appropriate classification algorithm, two main factors must be considered. One factor is that the classifier has to be capable of being trained incrementally. This is so that the classifier can be trained upon a stream, and simultaneously still provide valid classifications. This is essential, as some classifiers require parsing the entire data set before they are able to provide classifications, making them unsuitable for working with data stream. The other factor to consider is that of probabilities: it has been decided that it would be suitable for the classifier to return the probabilities of the hashtag recommendations it suggests (as seen in Functional Requirement 8).

These considerations have led to the choice of a Naive Bayes classifier. Naive Bayes is built upon Bayes' Theorem, which explains how to derive the probability of an event happening given that another event has occurred:

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)} \tag{4.1}$$

This can be expanded to give the probability of a classification based upon the presence of a set of features by making a **naive assumption**: that the presence of a particular feature is totally unrelated and unaffected by the presence (or lack) of any other feature. By making this assumption, we can derive the probabilistic model for a Naive Bayes classifier:

$$P(C|F_1, \ldots, F_n) = \frac{P(F_1, \ldots, F_n|C) \cdot P(C)}{P(F_1, \cdots, F_n)} \tag{4.2}$$

Or, in plain English terminology:

$$P(\text{Hashtag}|\text{Features}) = \frac{P(\text{Features}|\text{Hashtag}) \cdot P(\text{Hashtag})}{P(\text{Features})} \tag{4.3}$$

4.4 Data Features

As seen in the equations above, to predict hashtags for a tweet it is necessary to transform that tweet into a set of features. There are a number of approaches to this, and not all are limited to the text of the tweet. For instance, it could be possible to extract features from the metadata that is available with each tweet, such as date and time, or the location that the tweet was posted from. However, these are continuous variables and therefore present the extra challenge of discretisation before they can be used in a Naive Bayes classifier.

To reduce complexity, the feature extraction in this project will be based solely upon the actual text that makes up the tweets themselves.

4.5   User Interface

Whilst the primary focus of this project is the creation of the tool to provide hashtag suggestions and query expansions (the server), a prototype interface will also be developed. This interface will enable users to log in to Twitter, post tweets using hashtag suggestions, and search Twitter using query expansion.

Figure 4.3(a) shows the design of the interface for creating a tweet. It follows conventional HCI principles, including displaying the site name and navigation in the top left, and having the user's username and user-oriented controls in the top right. There is a single box for the user to type their tweet message into, and as they are typing into this box, relevant hashtag suggestions will appear underneath. Clicking on one of these hashtags will insert it at the end of the text in the box.

Figure 4.3(b) shows the design of the interface for searching Twitter. It follows the same HCI guidelines as most other search results pages, and keeps the same layout as the tweet composition page (and indeed, all other pages in the client interface).

(a) Tweet composition page.

(b) The search results page.

Figure 4.3: Wireframe designs of the prototype interface.

Chapter 5

Implementation

This chapter explains how the system has been implemented, and the decisions that were made during the process. It also shows any difficulties that were encountered, and how they were dealt with.

5.1 Tools & Technologies

The first decision to make when implementing the system was to decide upon the programming languages, technologies and libraries that would be used.

Python 2.7 was chosen as the language of choice for the majority of the project, including both the server and the content management system for the client. Python is a widely used high-level programming language whose syntax emphasises code readability and maintainability. Most importantly, it has a strong developer community behind it, resulting in a large collection of excellent third-party libraries, including the following:

Django[1] offers a simple and flexible way of following the traditional MVC (Model-View-Controller) programming paradigm. This made it easy to develop the client system, by providing a structured way to keep the authentication and page generation separate from the layout and design of the page. Django also offers a pleasant interface to handling web requests.

Flask[2] and **Tornado**[3] are web application frameworks made with Python that make it simple to create efficient and scalable web APIs and WSGI applications. They are used to create and handle the HTTP endpoints for the server.

Twython[4] is a thin but functional Python wrapper around the Twitter API that makes API calls easily available within Python. It also provides a set of methods that handle authentication with Twitter.

---

[1] www.djangoproject.com
[2] http://flask.pocoo.org
[3] www.tornadoweb.org
[4] http://github.com/ryanmcgrath/twython

Python 3 was not chosen for the project as it is still in its infancy. Many existing libraries written for Python have not yet been adapted for Python 3, and most operating systems or installers default to Python 2.7. To be successful, the system in this project has to be as easy to integrate and set up as possible — and therefore Python 2.7 was chosen.

The client interface itself was written with a combination of HTML, CSS and Javascript. In particular, the following tools and libraries were used:

Twitter Bootstrap[5] , a front-end framework that enables sleek and interactive interfaces with minimal customisation, allowing for easier and faster web development. It also has the bonus of being the UI toolkit that Twitter itself uses, and gives the client interface a look and feel similar to that of Twitter. Lastly, Bootstrap makes use of CSS3 to provide a full responsive interface, allowing the same web page to be viewed across all devices, regardless of the screen size — which helps to fulfil Non-Functional Requirement 6.

JQuery[6] , a Javascript library that contains a huge set of predefined functions and classes. It is designed to simplify client-side Javascript, and makes it much easier to manipulate the document object model (DOM) from within Javascript — it is used specifically within the client to perform asynchronous Javascript and XML requests (AJAX requests) to the server, and to provide live hashtag suggestions.

5.2 Spout: A Data Stream Processing Framework

To give the possibility of using either the live Twitter stream or a static file of tweets as the data source for the classifier, it was decided that the input data should be operated on as a stream. Doing so would allow identical processing of the tweet data, regardless of where the data came from.

Taking inspiration from a section of OpenIMAJ (Hare et al., 2011), a library with a fluent interface was created to enable such a processing paradigm. A fluent interface is the use of chaining method calls together to relay the instruction context of a subsequent call, to improve the readability of the code. At the heart of the library is the concept of a stream, which is simply a sequential (and not necessarily finite) collection of data items. The `Stream` class defines the basic operations that can be performed upon a data stream:

Mapping
The items in one stream can be mapped to another stream. This is done by applying a supplied `Function` object to each item in the input stream, to produce another output stream.

```
stream.map(Function)
```

---

[5]`http://getbootstrap.com`
[6]`www.jquery.com`

Filtering

The items in a stream can be filtered, so that the resultant stream only contains items that match a given criteria. This is done by using a supplied `Predicate` to test each item in the input stream, and copies it to the output stream if it passes the test criteria.

```
stream.filter(Predicate)
```

Processing (Consuming)

The items in a stream are used in some calculations or functionality that provides no further output to the stream. This is done by applying the supplied `Operation` to each item in the stream.

```
stream.for_each(Operation)
```

The source of the stream could be easily changed from the live Twitter Stream to a JSON file, or even to a plain text file. The interface is easily extendable, allowing for custom stream implementations to be created — all that is required is for the stream to output data items sequentially. A detailed class diagram of the core classes in Spout can be found in Appendix D.

The creation of this programming interface enabled a tweet processing pipeline to be constructed. The use of Spout has reduced the complexity of the pipeline, and makes the code much more readable, as seen in the code snippet in Listing 1 taken from the training code for the classifier.

```
twitter = TweetStream(QueueBufferedQueue(3)))
twiter.connect()

twitter
    .filter(TweetsWithHashtagsPredicate())
    .filter(TweetsInEnglishPredicate())
    .filter(NoRetweetsPredicate())
    .map(TokeniseTweetFunction())
    .for_each(TrainOperation(classifier)
```

Listing 1: Code snippet demonstrating classifier training using Spout stream processing.

Another feature built into Spout that can be seen in Listing 1 is the use of a `BufferedQueue` object when creating the data stream. Processing data from a live data source can present a common issue: the consumer-producer problem. This is where the input data source (in this case, the live Twitter stream) produces data faster than the system can process it, or vice-versa. Spout's `BufferedQueue` object defines a thread-safe queue of an arbitrary size that when full, discards any extra items that are offered. Similarly, if the queue is empty and another item is requested, the queue blocks the process asking for an item until another item is available. Thus, a trivial solution to the consumer-producer problem is presented, by filtering the input data source through a `BufferedQueue` before processing it.

5.2.1 Publication

Spout has been developed in an abstract, isolated fashion, meaning that it is distinct from the hashtag suggestion system presented in this project. This means that Spout could be useful in a number of scenarios other than Twitter processing. Because of this, the full source code for Spout has been made available independently from this project on GitHub[7].

In addition to open-sourcing Spout, it has also been made available on the Python Package Index (PyPI)[8]. This has been a huge success, with approximately 3000 recorded downloads at the time of writing.

5.3 Classification Server

5.3.1 Feature Extraction

Features are extracted from the tweets by passing their text through the 'twokeniser' system created by O'Connor et al. (2010).[9] This tool matches the text against a set of complex regular expressions to split the text up into a collection of word-like objects. It is a more involved process than splitting on whitespace alone however, as the tool correctly recognises and preserves complex structures such as urls and emoticons. The presence of each of the outputted tokens can then be used as a feature input for the classifier.

```
>>> tokeniser = TokeniseTweetFunction()
>>> tweet = "This is a #test! :-) http://url.com"
>>> tokeniser.apply(tweet)
['this', 'is', 'a', '#test', '!', ':-)', 'http://url.com']
```

Listing 2: A demonstration of the 'twokeniser' tool.

5.3.2 Training

To perform the calculations necessary for a Naive Bayes classifier, the server must store information about the tweets it has been trained on. It does this by updating a number of dictionaries stored in memory:

- A count of the total number of times each hashtag has been seen
  `hc = { hashtag : count }`

- A count of the number of times each token appears with a given hashtag
  `htc = { hashtag : { token : count } }`

---

[7]http://github.com/daviesjamie/spout
[8]https://pypi.python.org/pypi/spout/
[9]Available online at http://github.com/brendano/tweetmotif

- A count of the total number of times each token has been seen
  `tc = { token : count }`

- A count of the number of times each hashtag appears with a given token
  `thc = { token : { hashtag : count } }`

It is important to note here that the dictionaries simply store the count of each feature, and can therefore be quickly and easily incremented independently of the other features. Traditionally, Naive Bayes stores the probabilities of each feature — but doing this here would result in all of the probabilities having to be recalculated every time a new feature was processed. This would be a highly inefficient process, and so the probabilities are instead calculated on demand when needed.

In addition to this, the classifier also stores the total number of tweets and the total number of hashtag instances it has been trained on.

To update these internal models, the training function takes in a collection of features (tokens) and ensures that both hashtags and non-hashtag tokens are present. If this is the case, the models are incremented by looping through the tokens and adjusting the dictionaries.

A detailed class diagram of the `Classifier` class can be found in Appendix D.

5.3.3 Providing Recommendations

To classify a tweet, the Naive Bayes probability model is used to calculate the probability of every hashtag known to the classifier applying to the tweet. These probabilities are then sorted, and the top M probabilities (and their respective hashtags) are supplied as recommendations for the tweet.

The dictionaries and variables discussed above can be integrated into the Naive Bayes model to give a formula for calculating the probability of a hashtag given a set of tokens:

$$P(\text{Hashtag}|\text{Tokens}) = \frac{P(\text{Tokens}|\text{Hashtag}) \cdot P(\text{Hashtag})}{P(\text{Tokens})} \tag{5.1}$$

$$= \frac{\left(\prod_{i=1}^{N} \frac{\texttt{thc}[\text{Token}_i][\text{Hashtag}]}{\texttt{hc}[\text{Hashtag}]}\right) \cdot \frac{\texttt{hc}[\text{Hashtag}]}{\texttt{hashtag\_total}}}{\left(\prod_{j=1}^{N} \frac{\texttt{tc}[\text{Token}_j]}{\texttt{token\_total}}\right)} \tag{5.2}$$

Where:

- There are N tokens.
- `hashtag_total` is the total number of hashtags seen.
- `token_total` is the total number of non-hashtag tokens seen.

5.3.4   Optimisation

It can be seen in Equation 5.2 that the denominator of the equation is not related to the probability of the hashtag, and thus can be seen as a constant. This means that the denominator can be safely ignored to reduce the computational complexity of the equation:

$$P(\text{Hashtag}|\text{Tokens}) = \left(\prod_{i=1}^{N} \frac{\texttt{thc}[\text{Token}_i][\text{Hashtag}]}{\texttt{hc}[\text{Hashtag}]}\right) \cdot \frac{\texttt{hc}[\text{Hashtag}]}{\texttt{hashtag\_total}} \qquad (5.3)$$

Another issue with this approach to Naive Bayes is that when there is very little data for a given hashtag or token, the probability can become heavily biased. For example, if a token has only been seen once with a hashtag, then that token will have a 100% probability of being used in conjunction with that hashtag. This effect was overcome with a weighting function, $P*$, that adds a small bias towards an assumed probability:

$$P * (\text{Hashtag}|\text{Token}) = \frac{(\texttt{weight} \cdot \texttt{ap}) + (\texttt{tc}[\text{Token}] \cdot P(\text{Hashtag}|\text{Token}))}{\texttt{weight} + \texttt{tc}[\text{Token}]} \qquad (5.4)$$

Where:

- `weight` is the weighting of the bias, measured in number of tokens.

- `ap` is the assumed probability.

- $P(\text{Hashtag}|\text{Token})$ is the standard probability calculated using the method described in Equation 5.3.

Finally, the last optimisation made to the classification system was to filter the input data in an attempt to purify the data being trained upon. Using the stream processing framework presented in Spout (Section 5.2), it was possible to directly filter the tweets themselves upon a set of predicates, including removing all retweeted tweets[10], tweets that didn't contain any hashtags, and tweets that were not in English.

Another set of filters was configured to work upon the data from the tokenisation of the tweets including whether to allow url tokens, whether to allow username tokens, whether to allow punctuation tokens, and whether to use stop word removal. Removing stop words is a widely used technique in the field of natural language processing which checks input against a predefined list of words, and removes any occurrences of those words. This is typically used to remove words which have a very high frequency in text or speech. For a full list of the stop words used, see Appendix C. Experiments were also performed using another filter that uses the text of the hashtags in a tweet as extra tokens.

---

[10]A retweet is a re-posting of someone else's tweet.

5.3.5 Search Query Expansion

Due to the structure of the internal dictionaries inside the classifier, creating a query expansion is simple. The `htc` dictionary defines the relationship between hashtags and tokens — therefore to get a set of tokens relevant to a given hashtag, the tokens listed under `htc`[Hashtag] are sorted by their counts and then the top N are returned.

5.3.6 Asynchronous Access

To enable asynchronous training and classification function calls within the classifier, it became necessary to use a mutex lock to guard access to the internal models. Any function that required accessing or updating the models would have to acquire the lock first, and would wait until the lock became available if necessary. This approach meant that multiple threads could train the classifier and ask for hashtag recommendations simultaneously.

Unfortunately, this approach revealed a fundamental flaw with the Python thread execution model: regardless of how many threads are declared and executed within a piece of code, there will always be only a single thread executing at any given time. Python will automatically share execution time between active threads, and protects threads from interfering with each other through the global interpreter lock (GIL). In the case of the server system presented in this project, the GIL becomes a bottleneck that prevents the system from taking full advantage of multiprocessor systems.

The solution to this issue was to use operating system level processes in the place of threads. This gives each process a separate interpreter, and therefore a separate GIL and a separate thread of execution — achieving true parallel processing. However, this raised a new issue: each process has its own protected memory space, which therefore makes sharing complex data structures between processes difficult.

To successfully share the classifier object between processes, and not have each process operating on a separate instance, a proxy object is used. Extending the `BaseManager` class from the `multiprocessing`[11] library, the proxy object exposes a selection of the classifier methods to all of the processes in a virtual classifier object. The manager itself runs a server process, with which the other processes communicate via the virtual classifier object. The manager then relays the function calls upon the proxy object to a concrete instantiation of the classifier.

5.3.7 Creating a RESTful API

To enable clients to connect to the classifier and send recommendation or query expansion requests, the system must be available through a web interface. To achieve this, function calls to the classifier proxy object were wrapped into a series of web endpoints, using the Flask library. They make use of `POST` and `GET` data sent with

---

[11] `http://docs.python.org/2/library/multiprocessing.html`

the requests, and send all data output from the classifier back to the origin of the request in JSON format.

For a full list of the API endpoints used, please see Appendix E.

5.4 Client

The classification server is designed with an open REST API (see Appendix E), which creates the opportunity for a myriad of different clients to use the recommendation system. However, for the purposes of this project, a simple interface was created to demonstrate how the classification server can be accessed. For screenshots of the client interface, please see Appendix F.

The interface itself was built using Twitter Bootstrap to implement the wireframe designs in Section 4.5.

Django was used to handle the structure of the pages within the client, with each separate page being defined in its own `View`. Query expansions are acquired through a Django `View` that sends a `POST` request to the classification server, and search results through a `GET` request to the Twitter Search API.

To achieve instant hashtag suggestions as the user types a tweet, a custom JQuery function was used. This monitors the input form field for any changes, and sends an AJAX request to the classifier server when any changes are detected. The JSON object that is returned from the AJAX request is then parsed and displayed appropriately in the suggestions area on the page. When the user has completed and submitted the tweet, another Django view is used to process the input and send it to the Twitter API, thus posting the tweet.

Django views for both the popular hashtags and tokens use the `urllib`[12] library to fetch the relevant information from the classifier REST API, before embedding this data in a HTML page. The tag clouds were then generated using the `jquery.tagcloud.js`[13] JQuery plugin.

5.4.1 Authentication

Twitter uses OAuth to handle the authentication of users through third-party applications. OAuth is an open authentication protocol that allows users to approve applications to act on their behalf without sharing their password.[14]

A custom extension of the Django authentication module was created to use the Twitter OAuth API to authenticate users. Using the appropriate authentication methods exposed through the Twython library, users were created in Django that stored the OAuth tokens for each user after they had approved the client application to access their account. This then enabled use of the useful user library from

---

[12]`http://docs.python.org/2.7/library/urllib2.html`
[13]`http://github.com/addywaddy/jquery.tagcloud.js/`
[14]`http://dev.twitter.com/docs/auth/oauth/faq`

within Django, but also stored the OAuth tokens necessary for communication with Twitter.
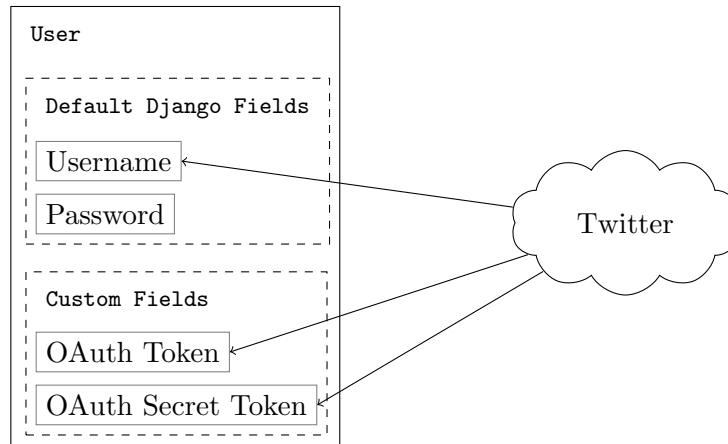
Figure 5.1: The components of a client `user` object.

5.5 Deployment

Due to the limited resources available for this project, the client and the server were both hosted on a virtual machine (VM) on the University of Southampton network. Separate instances of Tornado were used to serve the client and server on different ports to the local host only. Nginx[15] was then used to proxy outside requests from port 80 (the default HTTP port) to the correct ports on the local machine for the client and server. Additionally, Nginx was used to serve all static files, such as CSS or Javascript files. This approach not only significantly boosted the performance of the system, but also made it more secure and accessible.

It is intended, however, that in a real deployment scenario the client and server would be hosted on separate machines.
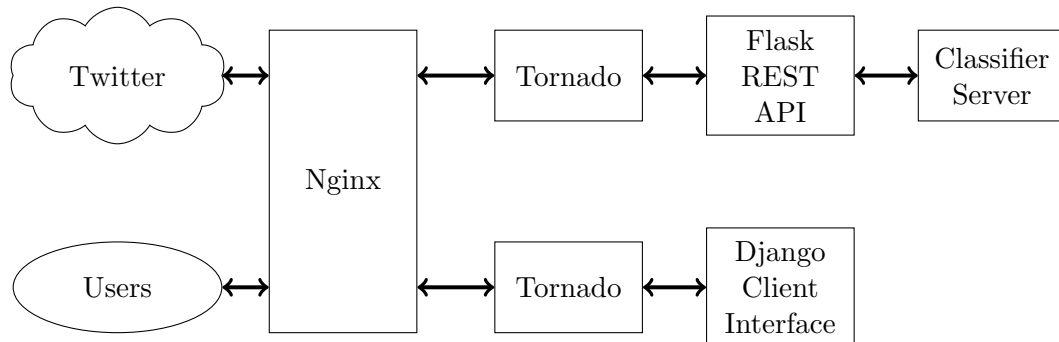
Figure 5.2: The deployment configuration of the project.

---

[15]www.nginx.org

Chapter 6

Testing Strategy

The system presented in this project was tested throughout the development process, as well as after the implementation was complete.

White-box testing was performed whilst creating every function, class and module in the system. This ensured that the inner structures and workings of the functions and objects were correct and performed as they should. Differing inputs were chosen for each test to exercise different paths through the code and ensure full code coverage. Maximising test code coverage is especially important within Python, as unlike traditional compiled languages, errors in the interpreted code are not found until the particular lines they are on are executed.

6.1   Unit Testing

The Python tool Nose[1] was used to create and execute unit tests for each of the separate units within the classifier server and Spout. These unit tests were written using black-box testing techniques to examine the functionality of the units without examining their internal structures and workings. For a full list of unit tests performed, please see Appendix A.

6.2   Integration Testing

Integration testing techniques were used to ensure that when the individual units were combined, they gave the expected results.

To test the REST API of the classifier server, tests were performed using the Unix `cURL` utility. This allows for easy execution of both `GET` and `POST` requests to the server by requiring minimal set up and configuration and printing all received data to the console.

---

[1]`https://pypi.python.org/pypi/nose/`

Testing the integration of the classifier components was a more involved task, requiring several short test scripts to be executed using specific data items chosen to exercise boundary cases. These tests were often executed interactively inside IPython[2]. IPython is an interactive Python shell that builds upon the default shell available with all Python installations. It offers enhanced functionality, including interactive examination of the data held within objects in a Matlab-style REPL (read, eval, print loop) environment. This made it possible to quickly pass data through a series of components and interactively examine the state of the components to ensure correctness of the code.

6.3   System Testing

To test the entire classification server system as a whole, there are two areas that need testing: the hashtag recommendation and the search query expansion.

To test the hashtag recommendation, 500,000 English, non-retweeted tweets with hashtags were collected from the live Twitter sample stream and stored as separate JSON objects in a static file (using the output tools in Spout). 5,000 of those tweets were then randomly selected as the testing set, leaving 495,000 tweets as training data. The training data was then fed into the training function for the classifier, and then the classifier was asked to suggest hashtags for each of the tweets in the training data (with their actual hashtags stripped out). The suggested hashtags from the classifier were then compared with the actual hashtags present in the tweets, to give an impression of the accuracy of the classifier. This was performed for a number of different combinations of filters, and the results can be viewed in Appendix B, and are discussed in detail in Chapter 8.

As the search query expansion makes use of the Twitter API, statistically evaluating the system becomes very difficult (as the Twitter Search API returns the latest tweets). Overcoming this would require either a higher level of access to the Twitter API, or the construction of a local search engine that can operate over a static set of tweets — both of which are unfortunately outside the scope of this project.

6.4   User Testing

The primary scope of this project was the creation and evaluation of the classification server tool, and therefore, the client interface presented is intended only for demonstration purposes and for integration testing.

If the client was to be published in its own right however, then it would require its own testing. This would be performed through a set of test users using the system to perform a predefined set of operations, and then supplying feedback on the interface and usability of the client.

---

[2] `www.ipython.org`

Chapter 7

Project Management

7.1 Time Planning

As with any project of this size, good planning and organisation is key to a successful project. Careful time planning results in a reliable schedule for the project, ensuring that all necessary deadlines are met.

The work involved in the completion of the project was carefully scheduled to maximise effective use of time, and to ensure that the components of the system were created in a logical order. The full detailed plan of the project was recorded in a Gantt chart, which can be found in Appendix G. Some key points to note about the project schedule are:

- The project as a whole had a lighter workload throughout the first semester, due to all the other modules in that semester having a particularly high coursework load.

- There was no work scheduled for the Christmas holiday or for the weeks directly after, due to the final exams and courseworks for the other modules taking priority. This also provided a 'buffer' period to catch-up on the project schedule if necessary.

- There was a light workload scheduled for the Easter holiday, allowing for any over-runs the project schedule may have had. This also provided time for revision and preparation for the exams at the end of the second semester.

By comparing the Gantt charts for the proposed progression of the project, and the reflection of the actual time spent on the areas within the project (also found in Appendix G), it can be seen that the plan for the project was relatively accurate. There were, however, a few deviations from the plan. Firstly, the amount of coursework set by other University modules in the first semester was underestimated resulting in the implementation for the project not starting until after the January exams. In addition to this, the January exam period was a week longer than originally accounted for, and therefore the project schedule was further pushed back.

7.2   Risk Analysis

Every large-scale project has to face a number of uncertain factors that could impact the progress or functionality of the final result. It is important to identify risks at the start of a project and take any steps necessary to reduce the likelihood or impact of those risks. By estimating the likelihood (a probability) of a risk occurring and the impact (on a scale of 1-4) that it would have, it is possible to calculate risk priority. This is a useful metric, as it enables realisation of the biggest threats to the project. Listed in Table 7.1 are some of the risks that could have occurred during the project, the impact they would have had, and how they would have been mitigated.

7.3   Version Control

Git[1] is a distributed version control system that allows the user to create a repository that stores a complete history and full version tracking. It is not dependent on network access or a central server (unlike the once popular alternative, Subversion[2]).

This entire project, including all source code and documentation, has been stored in a Git repository since its creation. This means that if a change is made to the project that is later decided to be unwanted, it is possible to use the Git tool to revert the file (or the whole working directory) back to a previous version. This also provides a form of protection against accidental modifications or deletion of files.

In addition to a local repository, the project is also hosted remotely on GitHub[3], meaning that it is possible to clone the project repository onto any machine with Git installed. This helped to synchronise the project across multiple development machines, in addition providing an off-site backup of all completed work. This means that in case of a hardware failure or other unexpected issues, a fully up-to-date copy of the project is easily accessible.

---

[1] `www.git-scm.com`
[2] `http://subversion.apache.org`
[3] `www.github.com`

Risk	Likelihood (L)	Impact (I)	Consequence of Occurence	Mitigation Measures	Priority ($L \times I$)
Unrealistic goals	0.6	4	Unable to complete work required to fulfil original project requirements	Simplify functionality of project to ensure that a core subset of the goals and features could still be achieved, in coordination with the project supervisor	**2.4**
Unexpected workload from other University modules	0.7	3	Unable to continue following project schedule	Carefully planned schedule to balance workload. Frequently update project Gantt chart to monitor the progress and ensure project remains on track	**2.1**
Hardware failure, loss or theft	0.3	4	Loss of all completed work	Push all work in a cloud-based Git repository, therefore creating an off-site backup	**1.2**
Falling ill, or other health issues	0.5	2	Reduced productivity for an extended period of time	Follow project schedule closely, minimising the work to catch-up on. Also allow 'buffer' periods in schedule to allow time for catching-up	**1.0**
Twitter API becoming unavailable or charging for access	0.1	4	Unable to use Twitter streaming, searching or authentication facilities	Use static data from an alternative source, such as the University of Southampton Twitter Archiver[4], and adjust functionality accordingly	**0.4**

Table 7.1: Project risk analysis, in descending order of priority.

---

[4] http://tweets.soton.ac.uk/

Chapter 8

Evaluation

8.1 Requirements Verification

By comparing the implemented system with the formal requirements described in Chapter 3, it is possible to verify that the system meets the original goals of the project.

8.1.1 Functional Requirements

#	Requirement	Status	Comments
1	The system must allow the user to log in and publish their tweets using their Twitter account.	Pass	The client authenticates with Twitter to allow the user to post tweets.
2	The system must provide hashtag recommendations as the user is writing a tweet.	Pass	The client displays hashtag suggestions as the user types.
3	The system must perform a hashtag search through a large dataset of tweets and return all relevant tweets, including those that do not contain the search query.	Pass	Query expansion is provided to enable searching with the Twitter API.
4	The system must use information from a large dataset of tweets to generate a model representing each hashtag.	Pass	The internal models in the classifier count the occurrences of tokens within each hashtag, amongst other things.

#	Requirement	Status	Comments
5	The system must use only the data available through the Twitter API, and not make use of any other externally collected metrics, such as user preferences.	Pass	Only the textual content of the tweet itself is used.
6	The system must be able to compare tweets against its representational hashtag models.	Pass	The classifier can calculate the probabilities for all hashtags for a given tweet.
7	*Optional:* In place of a large dataset, the system must be able to use information from the live Twitter stream.	Pass	A stream processing library was created to enable use of the live Twitter stream. The final system constantly trains on the live stream, dealing with classification requests asynchronously.
8	*Optional:* The system must provide probabilities for how likely a hashtag is to be related to a tweet.	Partial	The system provides probabilities, but they are not accurate due to the optimisations made to the classifier.

Table 8.1: Verification of Functional Requirements from Section 3.2.1.

8.1.2   Non-Functional Requirements

#	Requirement	Status	Comments
1	The system must be accessible via a web interface.	Pass	The classifier server is accessible through a REST API, enabling any web-based client to connect (including the prototype client developed in this project).

#	Requirement	Status	Comments
2	The system must be responsive and easy to use.	Pass	The classifier server is quick to respond to all requests, enabling a responsive client.
3	The system must be able to perform searches quickly.	Pass	The server provides a quick query expansion tool, meaning the speed of the searches is of comparable speed to the Twitter API search.
4	The system must be able to make hashtag recommendations quickly.	Pass	The classifier has been optimised to provide quick recommendations.
5	*Optional:* The system must be able to produce visualisations to provide an easy way to interpret the hashtag recommendations/assignments.	Partial	The system is capable of producing tag-clouds for hashtags and tokens, but other visualisations could be implemented.
6	*Optional:* The system must be accessible via mobile web browsers.	Pass	All major processing is done on the classification server, enabling any client to connect to the system — including ones optimised for mobiles. The included client interface is mobile-friendly.

Table 8.2: Verification of Non-Functional Requirements from Section 3.2.2.

8.2 Experimental Results

Besides arbitrarily entering strings of text and looking at the recommendations made, it is difficult to gauge the accuracy of the classifier. For this reason, a large dataset of tweets was collected and used to train and test the classifier — giving an idea of how good the recommendations are. For details of how this test was set up refer to Section 6.3, and for the full test results refer to Appendix B.

A number of interesting observations were drawn from the results of this test. It revealed that approximately 30% of the time the *first* hashtag suggested was correct, when using any single one of the filters (with the exception of filtering out usernames, which caused a drop of approximately 4.5%). Unfortunately though, when the top
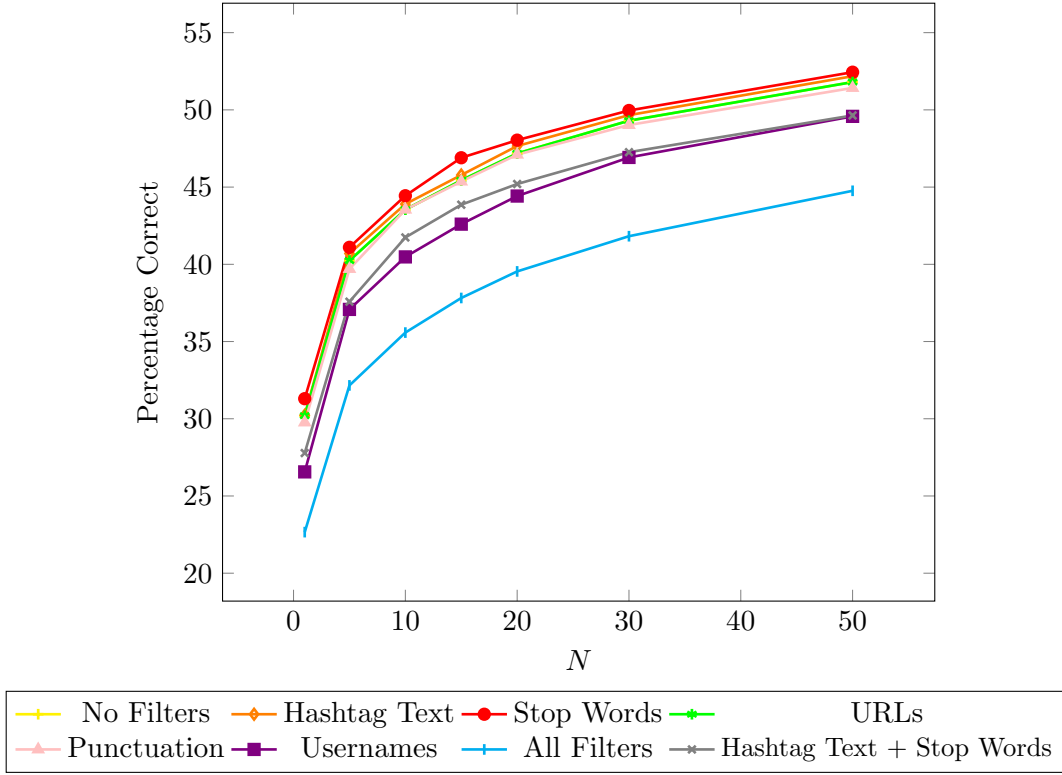
Figure 8.1: Percentage of **one or more** of N recommended hashtags being correct.

50 hashtag suggestions were used, it was discovered that only a maximum of 52% of the tweets were correctly assigned a hashtag — this is a number that was expected to be much higher. Some of this error could be attributed to hashtags that have not been seen before (as will always be the case on Twitter), but it is more likely that there is unfair biasing of some of the hashtags within the Naive Bayes classifier.

Surprisingly, when all of the filters were combined together, the accuracy of the classifier took a large drop, with the accuracy of the first suggestion falling to 22%. This is likely to be due to the removal of too many tokens, resulting in fewer tokens to actually classify. Further tests are needed to clarify this hypothesis, however.

An unsurprising observation is that the removal of URLs from the tweets made no impact on the accuracy of the classifications. This is accredited to the fact that most URLs within the training set would be unique, and therefore the probability of those URLs occurring would be neutral — resulting in a negligible impact to the overall probability calculated from the other tokens.

Impressively, when the test criteria is changed from the recommendation list having a single correct hashtag to containing all of the correct hashtags, the results only drop by approximately 10%. This is proof that the general concept of the classifier is working well: when a correct hashtag is recommended, so too are the related hashtags. The reason that this figure has taken a drop at all is attributed to some tweets containing multiple unrelated hashtags, which is difficult to predict.
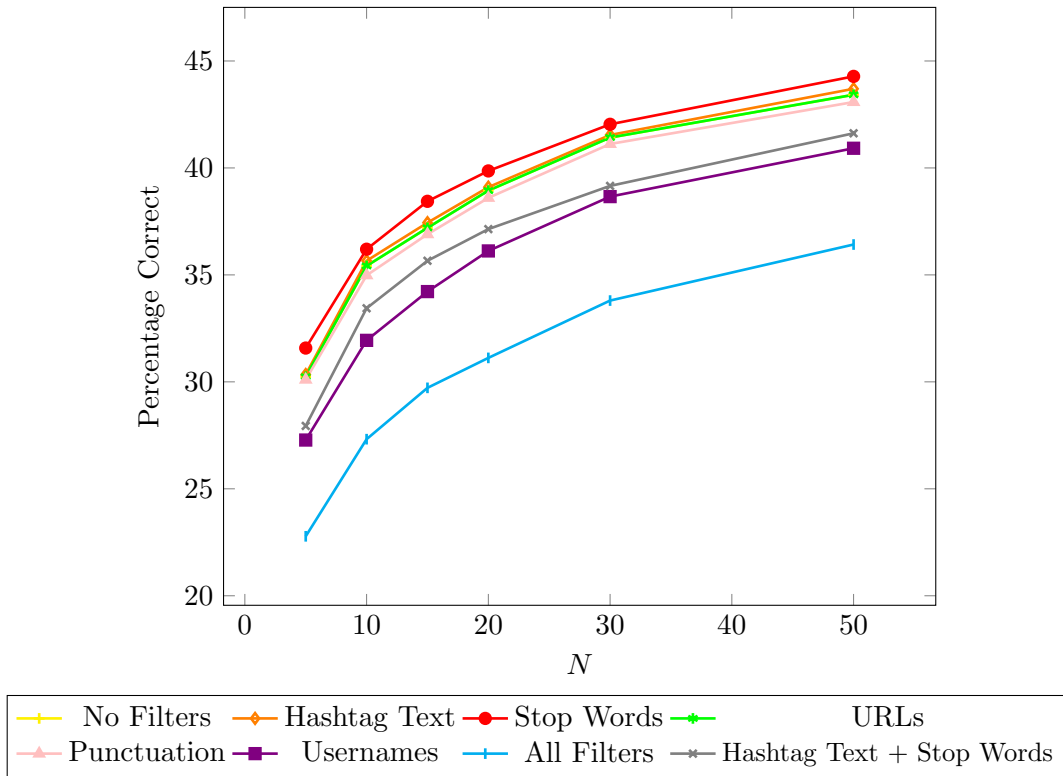
Figure 8.2: Percentage of **all** correct hashtags being present in N recommended hashtags.
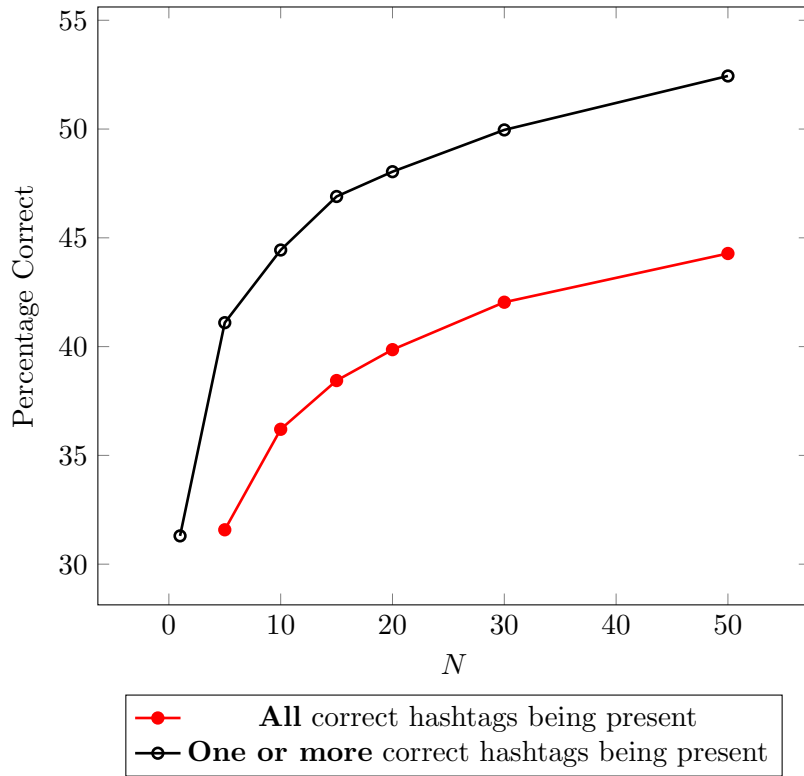

Figure 8.3: Percentage of hashtags correct when using the **Stop Words** filter.

Chapter 9

Conclusion

9.1 Reflection on the Project

9.1.1 Classification Server

Overall, the classification and recommendation system has been a success, but there are still things which could have been done differently. The results revealed by the experimental evaluation in Section 8.2 are truly just a starting point for the potential of the classifier. More work on the filtering of the classifier and the weighting of over-used or under-used hashtags is needed, for which there is no more time available in this project. The system excels at recommending hashtags for tweets about popular topics, but does not perform as adequately on less popular topics. More training data is required; allowing the classifier to train on data from a longer timespan would likely result in a significant increase in accuracy. This unfortunately is outside the scope of the project, as the time and resources necessary to gather such a large dataset are not available. In spite of this, the current system is capable of correctly predicting hashtags for between 30-50% of tweets, making it a functional tool.

9.1.2 Client

Although only created for demonstration purposes, the client has proved an invaluable tool during the optimisation of the classifier. It provides an environment in which tweets and query expansions can be quickly and easily evaluated, without having to delve into a console or interactive Python session. However, the current search functionality is unfortunately rather limited. It appears that Twitter performs their own query expansion within the API, so searching for a set of expanded query terms does not give the desired results. If this functionality was to be created again a separate tweet indexing and searching tool would be built, allowing for full control over the search process. With this improvement, it is easy to see the prototype client presented in this project being developed into a full interface, or even integrated into the Twitter platform itself.

9.1.3 Scalability

Due to the underlying client-server architecture, the system is highly scalable. Inside the classifier only counts of features are stored, and not entire tweets. This means that the system should be capable of training on millions, or potentially even billions, of tweets. The number of tweets it is possible to train upon is not infinite however, as eventually the integers used to store the counts will overflow. This could be combated by implementing an aging algorithm that reduces the counts on a periodical basis. As the number of unique tokens and hashtags seen by the system increases, it does unfortunately increase the time it takes to provide a classification. However, this task is currently only single-threaded, despite being highly parallelisable. Therefore the classification can be easily scaled by making use of the many cores available on modern server machines.

9.2 Achievement

This project has developed and presented an intelligent hashtag recommendation system for use with Twitter that makes it easier for users to compose tweets with relevant hashtags, as well as aiding search and navigation of Twitter as a whole. The presented technique of using a Naive Bayes classifier to predict hashtags has been shown to be a highly scalable approach, with promising results.

To aid with the handling of a live stream of Twitter data, a stream processing framework in Python was created. This uses a fluent interface to present a pleasant and readable infrastructure for processing streams of data — whether sourced directly from Twitter, or obtained from static files. This library has been published to the Python Package Index (PyPI), where it has been well received.

9.3 Future Work

The work presented within this project has provided a stable and scalable basis for future work. Much still needs to be done to improve the results of the classifier for it to be a fully usable tool. The filtering of the input tweets and tokens needs to be reassessed to improve the cases where there is the removal of too much information. Adjustments need to be made to the weighting of hashtags and tokens with very little occurrences, as well as those with massively popular counts.

The client presented in this project is simply a demonstration of the features available through the classification server. To become a usable tool in the 'real world,' an improved client would have to be designed and developed, and usability testing would have to be performed.

Lastly, the tests and experiments presented within this project have only been conducted on a dataset of 500,000 tweets. With more time and resources, this number could be increased to truly test both the scalability and accuracy of the classifier.

References

Cantador, I., Bellogín, A. and Vallet, D. (2010). 'Content-based Recommendation in Social Tagging Systems'. *Proceedings of the Fourth ACM Conference on Recommender Systems*. RecSys '10. Barcelona, Spain: ACM, pp. 237–240.

De Pessemier, T., Deryckere, T. and Martens, L. (2010). 'Extending the Bayesian classifier to a context-aware recommender system for mobile devices'. *Proceedings of the Fifth International Conference on Internet and Web Applications and Services*. ICIW 2010. Barcelona, Spain: IEEE Computer Society, pp. 242–247.

Gemmell, J., Schimoler, T., Ramezani, M. and Bamshad, M. (2009). 'Adapting K-Nearest Neighbor for Tag Recommendation in Folksonomies'. *Proceedings of the 7th Workshop on Intelligent Techniques for Web Personalization & Recommender Systems*. ITWP '09. Pasadena, CA, USA: CEUR Workshop Proceedings, pp. 51–62.

Goldberg, D., Nichols, D., Oki, B. M. and Terry, D. (1992). 'Using Collaborative Filtering to Weave an Information Tapestry'. *Communications of the ACM* 35.12, pp. 61–70.

Hare, J. S., Samangooei, S. and Dupplaw, D. P. (2011). 'OpenIMAJ and ImageTerrier: Java Libraries and Tools for Scalable Multimedia Analysis and Indexing of Images'. *Proceedings of the 19th ACM International Conference on Multimedia*. MM '11. Scottsdale, Arizona, USA: ACM, pp. 691–694.

Kywe, S. M., Hoang, T.-A., Lim, E.-P. and Zhu, F. (2012). 'On Recommending Hashtags in Twitter Networks'. *Proceedings of the 4th International Conference on Social Informatics*. SocInfo '12. Lausanne, Switzerland: Springer-Verlag, pp. 337–350.

Maron, M. E. and Kuhns, J. L. (1960). 'On Relevance, Probabilistic Indexing and Information Retrieval'. *Journal of the ACM (JACM)* 7.3, pp. 216–244.

Niwa, S., Doi, T. and Honiden, S. (2006). 'Web Page Recommender System Based on Folksonomy Mining for ITNG '06 Submissions'. *Proceedings of the Third International Conference on Information Technology: New Generations*. ITNG '06. Las Vegas, Nevada, USA: IEEE Computer Society, pp. 388–393.

O'Connor, B., Krieger, M. and Ahn, D. (2010). 'TweetMotif: Exploratory Search and Topic Summarization for Twitter'. *Proceedings of the Fourth International Conference on Weblogs and Social Media*. ICWSM 2010. Washington, DC, USA.

Oja, E. and Karhunen, J. (1985). 'On Stochastic Approximation of the Eigenvectors and Eigenvalues of the Expectation of a Random Matrix'. *Journal of Mathematical Analysis and Applications* 106.1, pp. 69–84.

Pazzani, M. J. and Billsus, D. (2007). 'The Adaptive Web'. Vol. 4321. Berlin, Heidelberg: Springer-Verlag. Chap. Content-based Recommendation Systems, pp. 325–341.

Salton, G. and Buckley, C. (1990). 'Improving Retrieval Performance by Relevance Feedback'. *Journal of the American Society for Information Science* 41.4, pp. 288–297.

Salton, G. and Buckley, C. (1988). 'Term-weighting Approaches in Automatic Text Retrieval'. *Information Processing and Management* 24.5, pp. 513–523.

Schafer, J. B., Konstan, J. A. and Riedl, J. (2001). 'E-Commerce Recommendation Applications'. *Data Mining and Knowledge Discovery* 5.1-2, pp. 115–153.

Schmitz, C., Hotho, A., Jäschke, R. and Stumme, G. (2006). 'Mining Association Rules in Folksonomies'. *Proceedings of the 2006 International Federation of Classification Societies Conference*. IFCS '06. Ljubljana, Slovenia: Springer, pp. 261–270.

Shepitsen, A., Gemmell, J., Mobasher, B. and Burke, R. (2008). 'Personalized Recommendation in Social Tagging Systems Using Hierarchical Clustering'. *Proceedings of the 2008 ACM Conference on Recommender Systems*. RecSys '08. Lausanne, Switzerland: ACM, pp. 259–266.

Utiyama, M. and Yamamoto, M. (2006). 'Relevance Feedback Models for Recommendation'. *Proceedings of the 2006 Conference on Empirical Methods in Natural Language Processing*. EMNLP '06. Sydney, Australia: Association for Computational Linguistics, pp. 449–456.

Xu, S., Bao, S., Fei, B., Su, Z. and Yu, Y. (2008). 'Exploring Folksonomy for Personalized Search'. *Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR '08. Singapore, Singapore: ACM, pp. 155–162.

Yang, L., Sun, T., Zhang, M. and Mei, Q. (2012). 'We Know What @You #Tag: Does the Dual Role Affect Hashtag Adoption?' *Proceedings of the 21st International Conference on World Wide Web*. WWW '12. Lyon, France: ACM, pp. 261–270.

Zangerle, E., Gassler, W. and Specht, G. (2011). 'Recommending #-Tags in Twitter'. *Proceedings of the Workshop on Semantic Adaptive Social Web*. SASWeb '11. Girona, Spain: CEUR Workshop Proceedings, pp. 67–78.

Appendices

A Unit Tests

The automated unit tests conducted throughout the project can be seen in the following tables. The only failure throughout the process was the `test_remaining_capacity` test, which failed because the `queue.qsize()` function from the `multiprocessing` library has not been implemented.

It is important to note that it was very difficult to write automatic tests for some of the functionality present in the project (for example, testing streaming tweets from Twitter), and therefore it was decided that these aspects would be tested manually.

TestClassifier	
Test Name	**Status**
`test_initialisation`	Pass
`test_training_fails_when_all_hashtags`	Pass
`test_training_fails_when_no_tokens`	Pass
`test_training_updates_models`	Pass
`test_training_updates_totals`	Pass
`test_use_hashtag_text_option`	Pass
`test_getters`	Pass
TestClassifierManager	
Test Name	**Status**
`test_methods_exists`	Pass
`test_models_are_protected`	Pass

The automated unit tests for the Classifier.

TestTweetsWithHashtagsPredicate	
Test Name	**Status**
test_accept	Pass
test_reject	Pass

TestTweetsInEnglishPredicate	
Test Name	**Status**
test_accept	Pass
test_reject	Pass

TestNoRetweetsPredicate	
Test Name	**Status**
test_accept	Pass
test_reject	Pass

The automated unit tests for the filters.

TestTokeniseFunctions	
Test Name	**Status**
test_tokenisation	Pass
test_case_insensitive	Pass
test_remove_punctuation	Pass
test_remove_stop_tokens	Pass
test_remove_usernames	Pass
test_remove_urls	Pass
test_remove_all_together	Pass

The automated unit tests for the Tokenisation functions.

TestQueueBufferedQueue	
Test Name	**Status**
`test_adding_to_queue`	Pass
`test_removing_from_queue`	Pass
`test_dropping_when_full`	Pass
`test_remaining_capacity`	Fail
TestUtils	
Test Name	**Status**
`test_true_predicate`	Pass
`test_false_predicate`	Pass
`test_pass_through_function`	Pass

The automated unit tests for the Spout utilities and data structures.

TestStream	
Test Name	**Status**
`test_testing_stream`	Pass
`test_filter_returns_filter_stream`	Pass
`test_map_returns_map_stream`	Pass
`test_for_each_returns_nothing`	Pass
`test_filter_is_applied`	Pass
`test_map_is_applied`	Pass
`test_operation_is_applied`	Pass

The automated unit tests for the Spout streams.

B Test Results

Filters					N						
Hashtag Text	**Stop Words**	**URLs**	**Punctuation**	**Usernames**	**1**	**5**	**10**	**15**	**20**	**30**	**50**
					30.20	40.24	43.54	45.44	47.18	49.30	51.80
•					30.22	40.70	43.90	45.78	47.66	49.66	52.18
	•				31.30	41.10	44.44	46.90	48.04	49.96	52.44
		•			30.20	40.24	43.54	45.44	47.18	49.30	51.80
			•		29.76	39.70	43.54	45.36	47.10	49.02	51.42
				•	26.56	37.08	40.48	42.60	44.42	46.92	49.58
•	•	•	•	•	22.66	32.16	35.58	37.82	39.54	41.82	44.76
•	•				27.78	37.58	41.74	43.86	45.20	47.26	49.64

Test results showing the percentage of **one or more** of N recommended hashtags being correct.

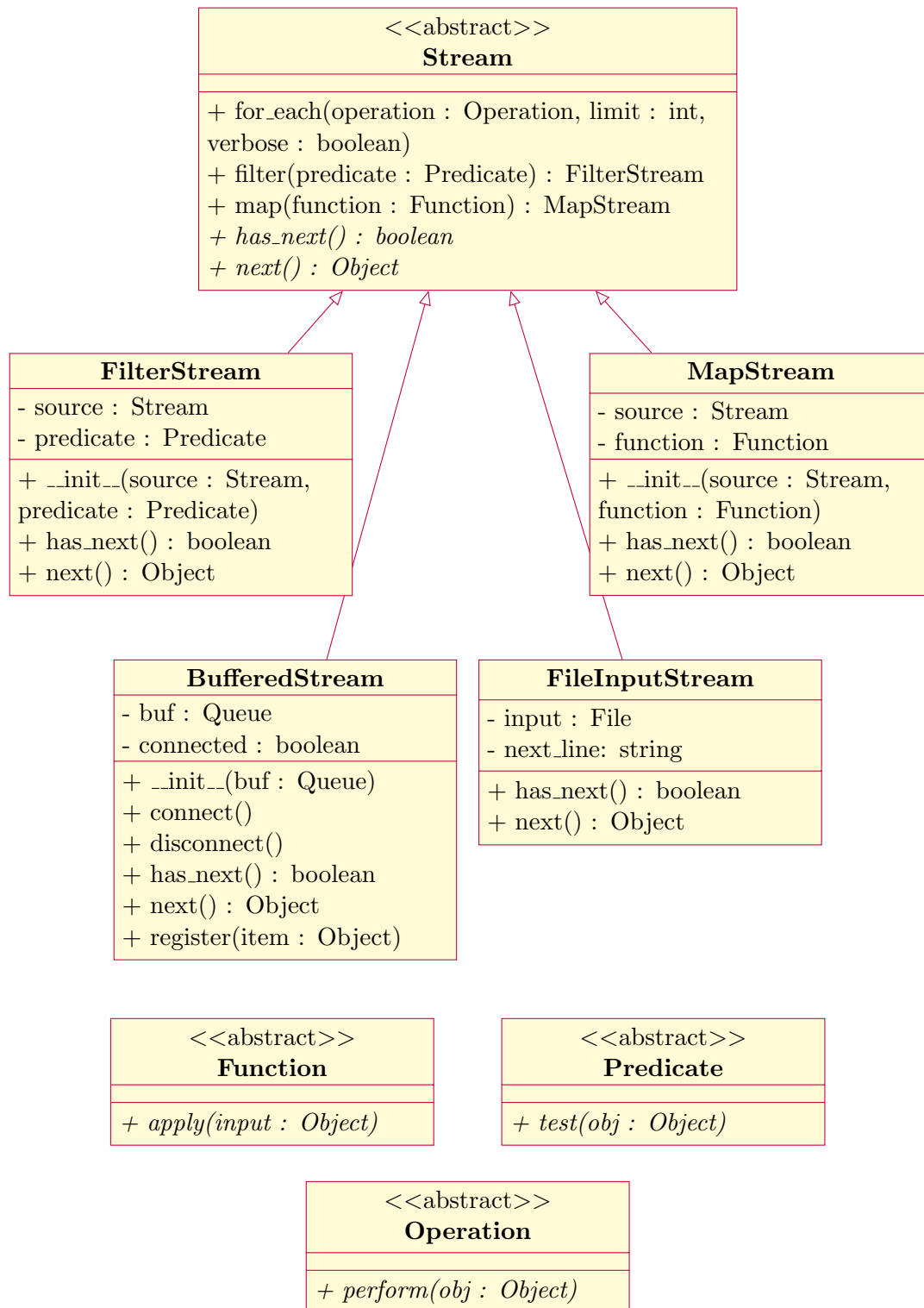| Filters					N					
Hashtag Text	**Stop Words**	**URLs**	**Punctuation**	**Usernames**	**5**	**10**	**15**	**20**	**30**	**50**
					30.24	35.42	37.20	38.94	41.42	43.42
•					30.34	35.66	37.44	39.10	41.54	43.70
	•				31.58	36.20	38.44	39.86	42.04	44.28
		•			30.24	35.42	37.20	38.94	41.42	43.42
			•		30.10	34.98	36.90	38.60	41.12	43.08
				•	27.28	31.94	34.22	36.12	38.66	40.92
•	•	•	•	•	22.78	27.32	29.42	31.12	33.80	36.42
•	•				27.94	33.44	35.66	37.14	39.16	41.62

Test results showing the percentage of **all** of the correct hashtags being present in N recommended hashtags.

C   Stop Word List

Shown below is the stop word list that was used to filter out high-frequency words from the input tokens to the classifier. Please see Subsection 5.3.4 for an explanation.

a	had	lets	that	well
about	has	me	thats	were
above	have	more	the	weve
after	having	most	their	were
again	he	my	theirs	what
against	he'd	myself	them	whats
all	he'll	no	themselves	when
am	hed	nor	then	whens
an	hell	not	there	where
and	hes	of	theres	wheres
any	her	off	these	which
are	here	on	they	while
as	heres	once	they'd	who
at	hers	only	they'll	whos
be	herself	or	they're	whom
because	him	other	they've	why
been	himself	ought	theyd	whys
before	his	our	theyll	with
being	how	ours	theyre	won't
below	hows	ourselves	theyve	would
between	i	out	this	you
both	i'd	over	those	you'd
but	i'll	own	through	you'll
by	i'm	rt	to	you're
could	i've	same	too	you've
did	id	she	under	youd
do	ill	she'd	until	youll
does	im	she'll	up	youre
doing	ive	shed	very	youve
down	if	shell	was	your
during	in	shes	we	yours
each	into	should	we'd	yourself
few	is	so	we'll	yourselves
for	it	some	we're	's
from	its	such	we've	
further	itself	than	wed	

D Class Diagrams

<>
Stream

+ for_each(operation : Operation, limit : int, verbose : boolean)
+ filter(predicate : Predicate) : FilterStream
+ map(function : Function) : MapStream
+ has_next() : boolean
+ next() : Object

FilterStream

- source : Stream
- predicate : Predicate

+ __init__(source : Stream, predicate : Predicate)
+ has_next() : boolean
+ next() : Object

MapStream

- source : Stream
- function : Function

+ __init__(source : Stream, function : Function)
+ has_next() : boolean
+ next() : Object

BufferedStream

- buf : Queue
- connected : boolean

+ __init__(buf : Queue)
+ connect()
+ disconnect()
+ has_next() : boolean
+ next() : Object
+ register(item : Object)

FileInputStream

- input : File
- next_line: string

+ has_next() : boolean
+ next() : Object

<>
Function

+ apply(input : Object)

<>
Predicate

+ test(obj : Object)

<>
Operation

+ perform(obj : Object)

The **core** classes that make up the Spout library. The full code and all classes can be found at `http://github.com/daviesjamie/spout`.

Classifier
– hashtag_total : int
– hc : dict
– htc : dict
– lock : multiprocessing.Lock
– start_time: int
– tc : dict
– thc : dict
– tweet_total : int
– use_hashtag_text : boolean
+ __init__(use_hashtag_text : boolean)
+ classify(tweet : string, results : int) : list
+ get_hashtags(num : int) : list
+ get_hashtag_tokens(num : int) : list
+ get_memory_usage() : list
+ get_tokens(num : int) : list
+ get_token_hashtags(num : int) : list
+ get_totals() : list
+ get_unique_counts() : list
+ get_uptime() : datetime.timedelta
+ state_dump(fileprefix : string)
+ state_load(filename : string)
+ train(tweet_tokens : list)
– _catcount(hashtag : string) : int
– _docprob(tokens : list, hashtag : string) : float
– _fcount(token : string, hashtag : string) : int
– _fprob(token : string, hashtag : string) : float
– _hashtags() : list
– _prob(tokens : list, hashtag : string) : float
– _totalcount() : int

The `Classifier` class, used within the server to provide all recommendations.

E   REST API Interface

Listed in the following table are the endpoints (URLs) used for the for RESTful server API.

Endpoint	Method	Functionality	Parameters
/api/classify	POST	Classify a tweet to provide a list of hashtag suggestions	`text` — The text of the tweet to classify `results` — The number of classifications (recommendations) to provide (*optional*)
/api/status	GET	Return a JSON object with useful information and statistics about the server	
/api/hashtags	GET	Returns an ordered list of all hashtags and their counts	`num` — The number of hashtags to return (*optional*)
/api/tokens	GET	Returns an ordered list of all tokens and their counts	`num` — The number of tokens to return (*optional*)
/api/hashtag /<string:hashtag>	GET	Returns an ordered list of all tokens and their counts that have been seen with the given hashtag	`num` — The number of tokens to return (*optional*)
/api/token /<string:token>	GET	Returns an ordered list of all hashtags and their counts that have been seen with the given token	`num` — The number of hashtags to return (*optional*)

A list of the endpoints available through the RESTful API.
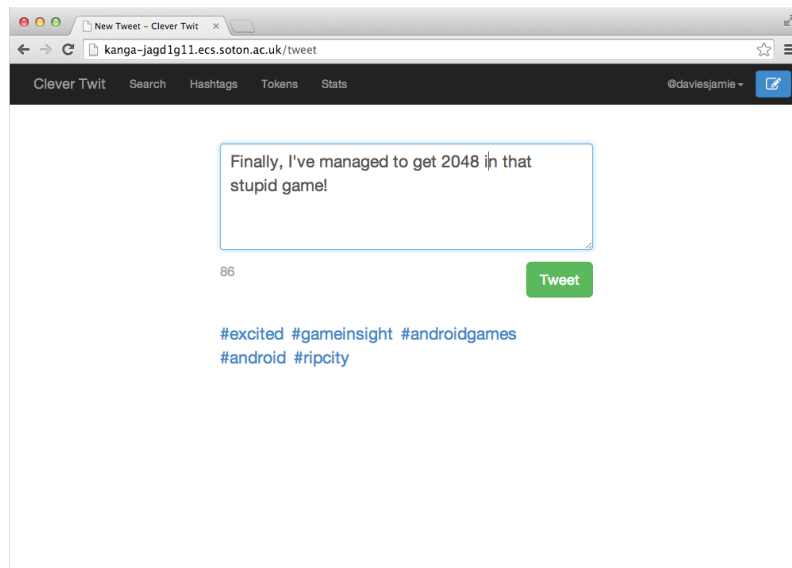
F   Client Interface Screenshots
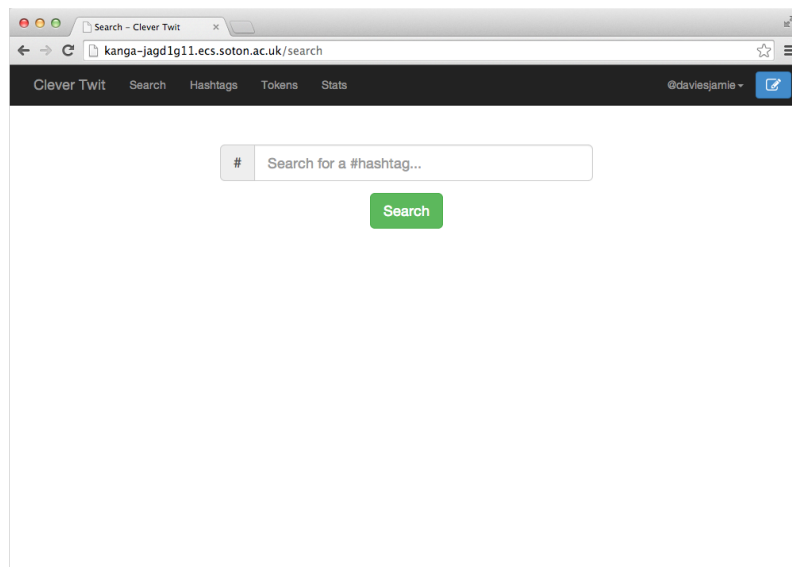

Figure 9.1: The new tweet page.
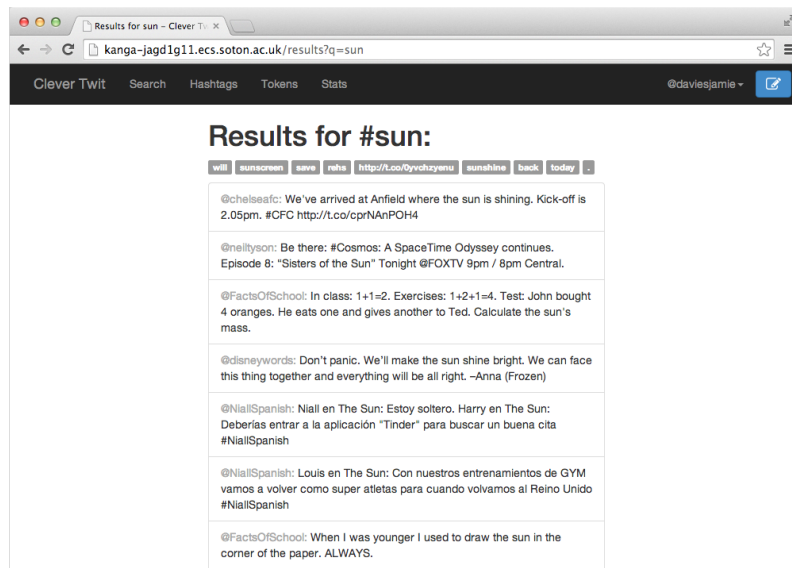

Figure 9.2: The search query page.

Figure 9.3: The search results page.
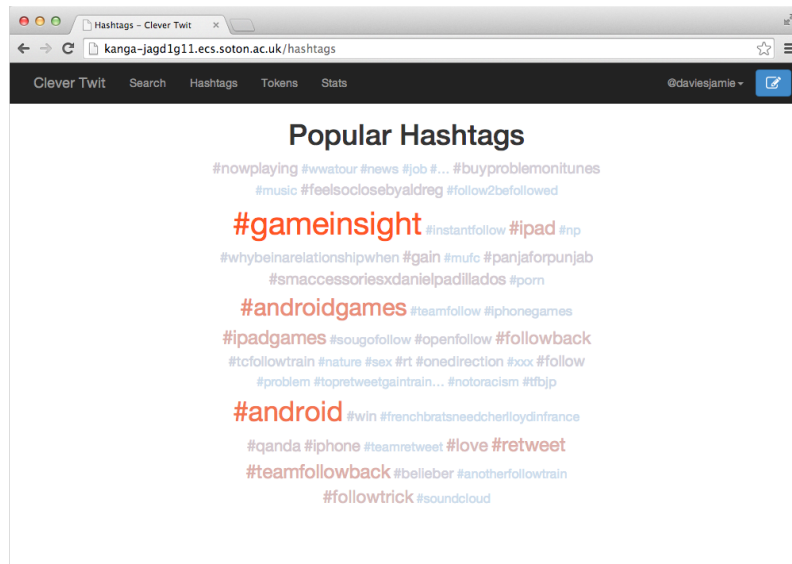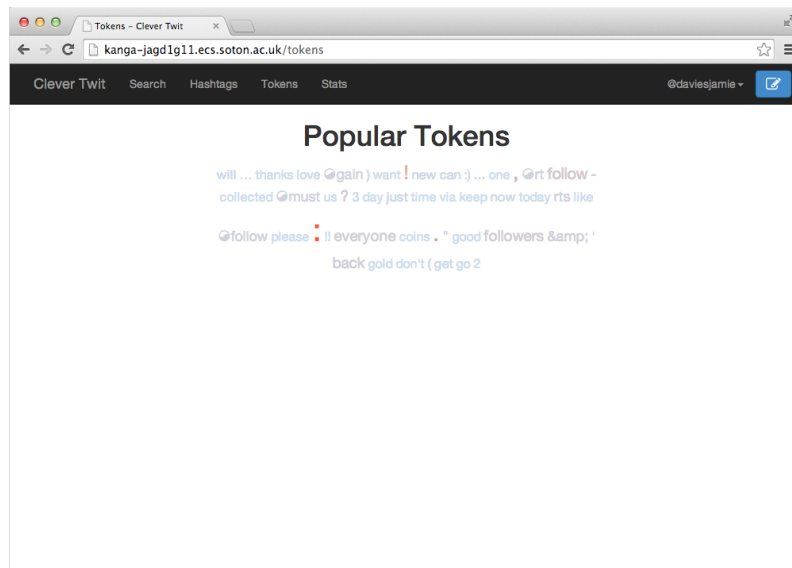

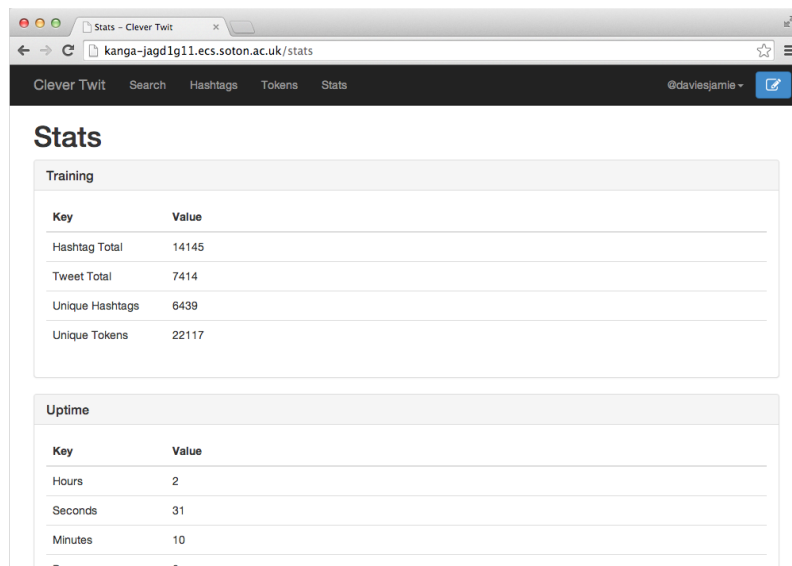Figure 9.4: The popular hashtags page.

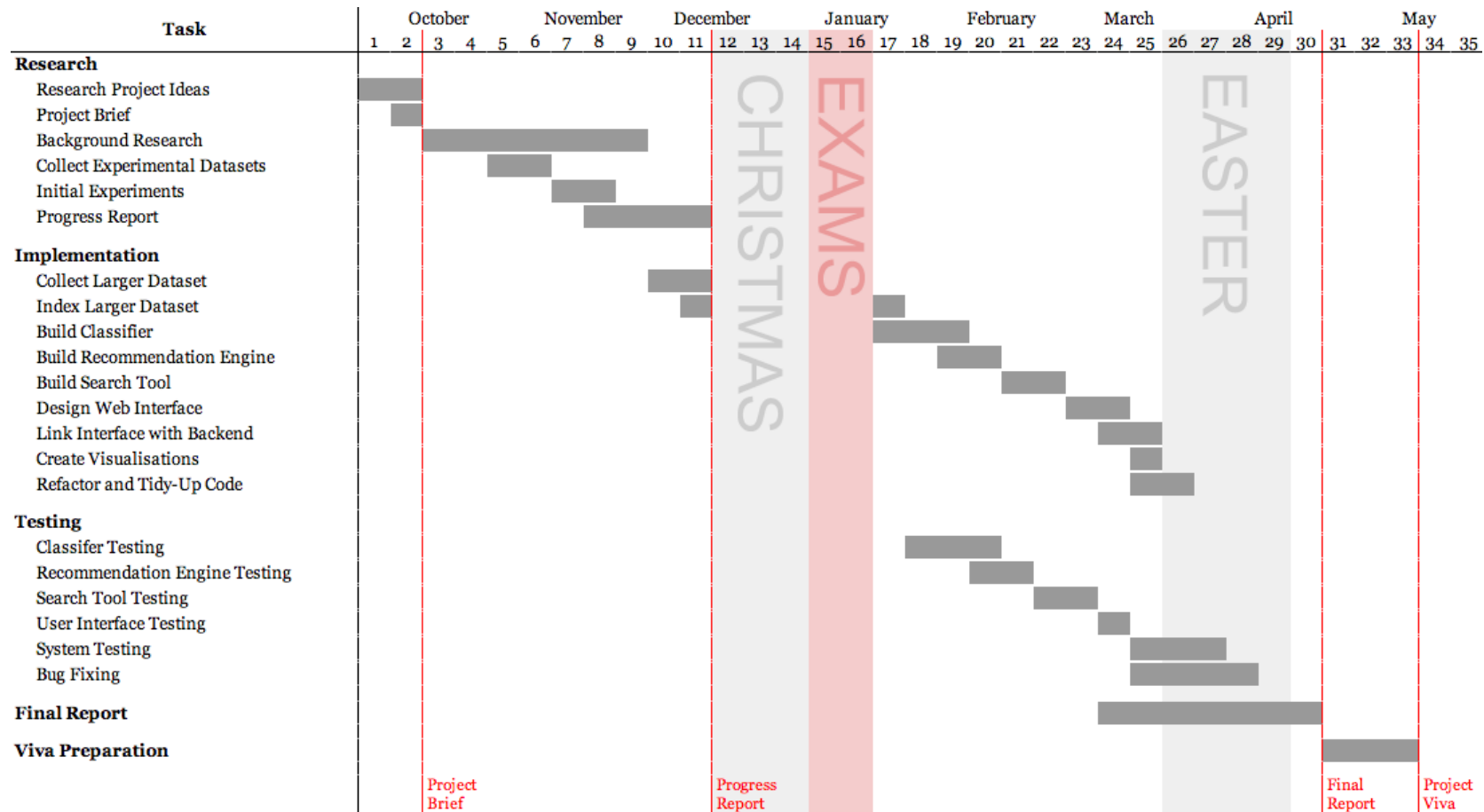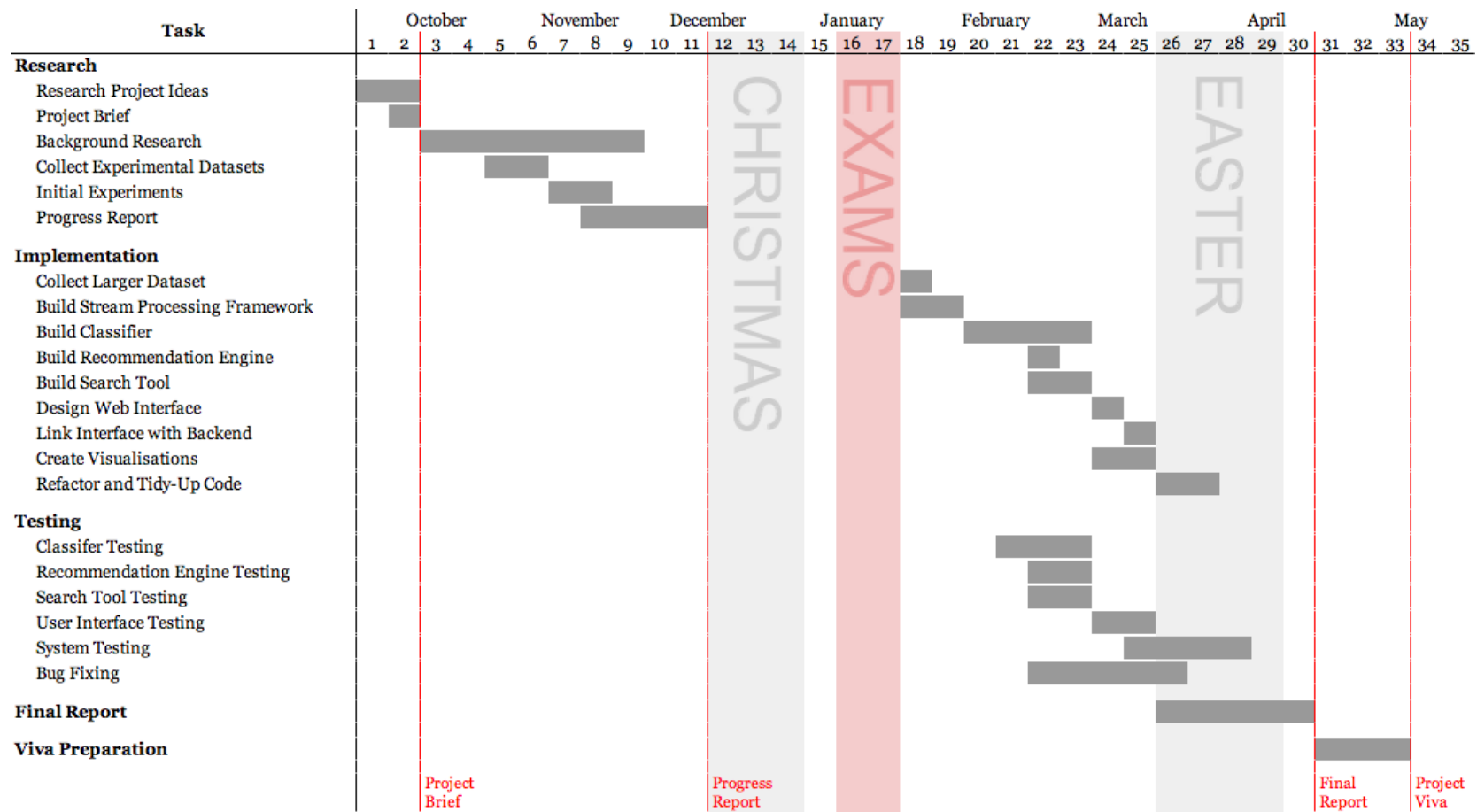Figure 9.5: The popular tokens page.

Figure 9.6: The classifier statistics page.

G   Gantt Charts

Task	October		November		December		January		February		March		April		May
	1 2	3 4 5	6 7 8 9	10 11	12 13 14	15 16	17 18 19	20 21 22 23	24 25	26 27	28 29 30	31 32	33	34 35	

Research
- Research Project Ideas
- Project Brief
- Background Research
- Collect Experimental Datasets
- Initial Experiments
- Progress Report

Implementation
- Collect Larger Dataset
- Index Larger Dataset
- Build Classifier
- Build Recommendation Engine
- Build Search Tool
- Design Web Interface
- Link Interface with Backend
- Create Visualisations
- Refactor and Tidy-Up Code

Testing
- Classifer Testing
- Recommendation Engine Testing
- Search Tool Testing
- User Interface Testing
- System Testing
- Bug Fixing

Final Report

Viva Preparation

CHRISTMAS   EXAMS   EASTER

Project Brief   Progress Report   Final Report   Project Viva

A Gantt chart showing the *scheduled* progression through the different aspects of the project.

A Gantt chart showing the *actual* progression through the different aspects of the project.

H Project Directory Listing

```
client
├── bower.json
├── clever_twit
│   ├── __init__.py
│   ├── admin.py
│   ├── models.py
│   ├── templates
│   │   ├── base.html
│   │   ├── cover.html
│   │   ├── default.html
│   │   ├── hashtag.html
│   │   ├── hashtags.html
│   │   ├── index.html
│   │   ├── results.html
│   │   ├── search.html
│   │   ├── stats.html
│   │   ├── token.html
│   │   ├── tokens.html
│   │   └── tweet.html
│   ├── tests.py
│   ├── urls.py
│   └── views.py
├── client
│   ├── __init__.py
│   ├── oauth.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
├── manage.py
├── oauth.json
├── server.py
├── static
│   └── css
│       └── styles.css
├── twython_django
│   ├── __init__.py
│   ├── models.py
│   ├── templates
│   │   └── tweets.html
│   ├── urls.py
│   └── views.py
├── nginx.conf
└── requirements.txt
```

```
server
├── __init__.py
├── classifier.py
├── classifier_manager.py
├── experiments
│   ├── experiment_harness.py
│   └── tweets.json
├── filters.py
├── oauth.json
├── oauth.py
├── output.py
├── server.py
├── stoptokens.txt
├── tests
│   ├── __init__.py
│   ├── test_classifier.py
│   ├── test_filters.py
│   └── test_tokeniser.py
├── tokeniser.py
└── twokenize
    ├── __init__.py
    ├── emoticons.py
    └── twokenize.py
```

```
spout
├── LICENSE.txt
├── README.md
├── setup.py
├── spout
│   ├── __init__.py
│   ├── outputs.py
│   ├── queues.py
│   ├── sources.py
│   ├── streams.py
│   ├── structs.py
│   └── utils.py
└── tests
    ├── __init__.py
    ├── test_queues.py
    ├── test_streams.py
    └── test_utils.py
```

I Original Project Brief

<center>Part III Individual Project Brief</center>

Enhanced Content Analysis and Information Retrieval Using Twitter Hashtags

<center>

Author: *Supervisor:*
Jamie Davies   Dr. Nicholas Gibbins
`jagd1g11@ecs.soton.ac.uk`   `nmg@ecs.soton.ac.uk`

</center>

Microblogging services such as Twitter[1] have become very popular in recent years. One of the key characterists of such services is the use of 'hashtags' — topical/categorical annotations provided by the authors of the posts (tweets) themselves. These hashtags have been proven to be useful and information-rich in previous microblogging research projects; from using them to crowd-source real-time event detection [2], to using them to train sentiment classifiers [1]. However due to their nature, there is a vast and diverse collection of hashtags for users to choose from, resulting in the information provided by hashtags not being as accurate or as complete as it could be.

For my project, I will create a system that aims to support and enrich the information provided by hashtags. I will use the latest machine-learning techniques to examine and classify the topics and concepts behind the hashtags and in doing so, be able to suggest suitable new hashtags for tweets that are relevant to their content. This will allow users to make a better choice of hashtags when writing their tweets, and therefore refine the information they provide. Furthermore, I will create an extension of this system that will provide a context-aware tweet search facility. This will enable a user to search for a particular hashtag, and instead of only returning tweets containing that hashtag (as current systems do), it will also provide tweets that are contextually relevant to the search term but do not contain that given hashtag.

Finally, I will test the success and importance of my research by implementing a recent research experiment that depends upon hashtags (such as those conducted by Davidov et al. [1] or Sakaki et al. [2]). I will compare the results of the experiment both with and without my enhanced system, and provide validation for the project.

References

[1] D. Davidov, O. Tsur, and A. Rappoport. Enhanced sentiment learning using twitter hashtags and smileys. In *Proceedings of the 23rd International Conference on Computational Linguistics: Posters*, COLING '10, pages 241–249, Stroudsburg, PA, USA, 2010. Association for Computational Linguistics.

[2] T. Sakaki, M. Okazaki, and Y. Matsuo. Earthquake shakes twitter users: real-time event detection by social sensors. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 851–860, New York, NY, USA, 2010. ACM.

---

[1]`http://www.twitter.com`