Todo list

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

Jamie Davies
24th April 2014

Enhanced Content Analysis and Information Retrieval Using Twitter Hashtags

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.

If users are presented with a choice of relevant hashtags when writing a tweet, they are more likely to publish tweets with accurate tag data. This project aims to create an intelligent hashtag recommendation tool to improve the quality of the information we can gain from hashtags. However, whilst such a system could improve future tweets, tweets that have already been published will remain untouched by the system. Thus, the system will be extended to also retrofit hashtags to published tweets — allowing for tweets to appear in search results for a particular hashtag even if they don't actually contain the hashtag in question.

Make shorter

Add more about scope of project

Contents

Acknowledgements

I would like to thank my supervisor, Dr. Nick Gibbins, for providing inspiration and guidance throughout the course of the project.

I would also like to thank Dr. Jonathon Hare and Dr. Sina Samangooei for their continued support and advice.

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.

1.1 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 1.1 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 and spread across several tags.

Another interesting observation is that of the times that different hashtags are used, which varies greatly from hashtag to hashtag. Some tags are used in tweets at a fairly

[1] `www.twitter.com`

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

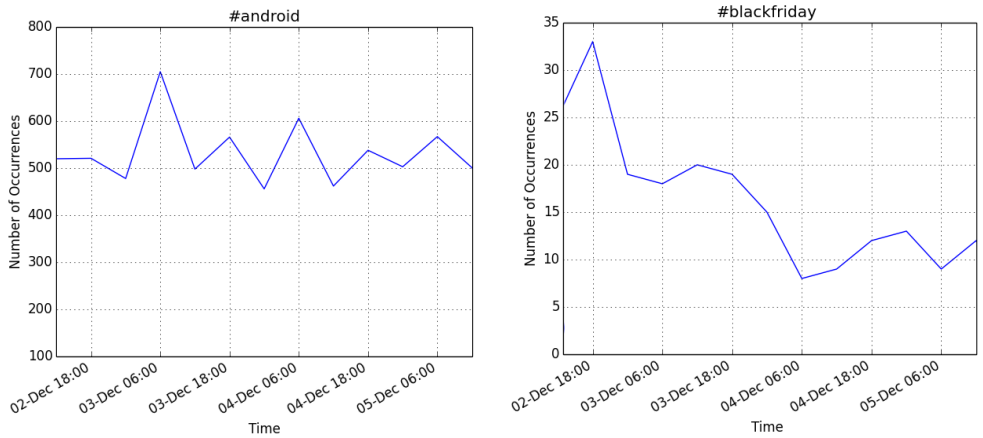
uniform rate (Fig. 1.1(a)), whilst others feature large spikes of usage over a short period of time (Fig. 1.1(b)). This demonstrates the communities behind the hashtags — rather than a hashtag being used at a particular time for a particular event, some hashtags are used consistently by users to join in on a large-scale conversation.

This data has shown that 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 tag redundancy can confuse users and fragment the true meaning behind the synonymous hashtags.

1.2 Project Goals

The goal of this project is to develop an intelligent hashtag recommendation system for use with Twitter. For example, when a user writes a tweet such as: "*The golden gate bridge looks so nice today! I LOVE SF in this weather!*", the system should recommend hashtags such as `#SanFrancisco`, `#California` or even `#sohappy`, but should not recommend hashtags such as `#ScienceFiction`, `#amreading` or `#fantasy`.

The objective is to make it easier for users to select appropriate meaningful 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 making



(a) #android (b) #blackfriday

Figure 1.1: Usage of hashtags over the period that the dataset was collected.

them more likely to publish tweets with accurate hashtag data.

Such a recommendation system would also need to aid users in the navigational and exploratory aspects of folksonomies. For example, when a user performs a search on Twitter for the hashtag `#BGT`, the system should display not only those tweets that contain `#BGT`, but also tweets that are relevant to 'Britain's Got Talent' but do not contain `#BGT`.

Therefore, another objective for this project is to make it easier for users to navigate and search the content available on Twitter, thus making the fragmented hashtag data already published to Twitter easier to search and navigate. The aim is that the same recommendation system used to suggest hashtags for content a user is creating could also be applied to a search query to provide an expanded query that can then be used to query Twitter.

Chapter 2

Background Literature & Related Work

The main design goals behind hashtags are 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]) to suggested purchases on retail sites (like on the front page of Amazon[3]), these systems are all personalised recommendation engines that take an individual user's preferences and use them to provide suggestions tailored to that user.

Add screenshots of existing recommendation services?

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 hasn't 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

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

items relevant to other items by comparing the details and descriptions of the items themselves. 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 incorporate relevance feedback techniques. Relevance feedback is a process that was originally designed for information retrieval, and works on the assumption that a user can not always correctly encapsulate into a query what it is they are searching for. It works by allowing a user to create an initial query to which an initial set of results is returned. Out of these initial results, the user can then mark certain results as relevant or irrelevant, and 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." 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.

Add more about K-Nearest Neighbour implementation?

Niwa et al. (2006) took a different approach to overcome the limitations of collaborative filtering. Using a modified version of the 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.

Add more TF-IDF implementation detail?

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

Add more about Naive Bayes implementation?

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 2.1) uses a non-personalised autocomplete tool to provide suggestions to the user. Whenever a hash symbol (#) is typed in the tweet composer, the system simply suggests hashtags starting with the

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

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.

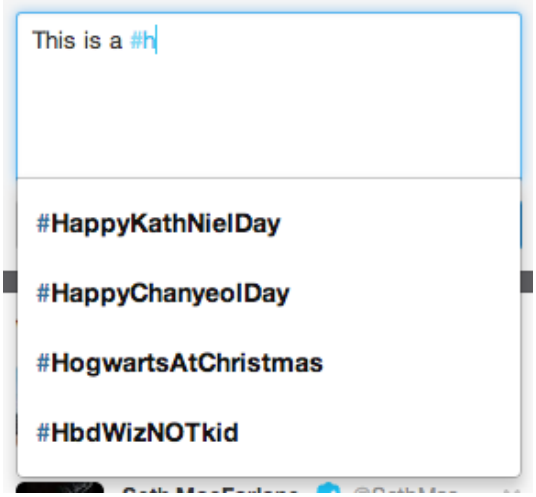


Figure 2.1: Twitter's current hashtag suggestion system.

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 how similar the tweets were to the original query, and returned as a list of suggestions to the user. A number of different ranking algorithms were tested, but this 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 user to user, with some users using the latest trending hashtags, other users only using a specific set or type of tag, and with some users barely using 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

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

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: connecting users together. Yang et al. (2012) took this duality into account when attempting to create a hashtag recommendation system by training a SVM (support vector machine) classifier with a variety of features taken from the tweet metadata to overcome the duality and suggest relevant hashtags.

Chapter 3

Analysis

As previously discussed in Chapter 1, 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 already existing fragmented data. This leads to the two main requirements the system in this project must fulfil:

- Users should be able to compose and publish tweets whilst suggesting 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 Chapter 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 it is this preference of users to use a particular hashtag that results in users incorrectly using hashtags in the first place — 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.

3.1 Requirements

From the broad requirements outlined above, a set of formal requirements can be inferred:

3.1.1 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 must be able to use information from the live Twitter stream.
8. *Optional:* The system must provide probabilities for how likely a hashtag is to be related to a tweet.

3.1.2 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 must be able to produce visualisations to provide an easy way to interpret the hashtag recommendations/assignments.
6. *Optional:* The system must 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 (Figure 4.1) has been chosen for the system.

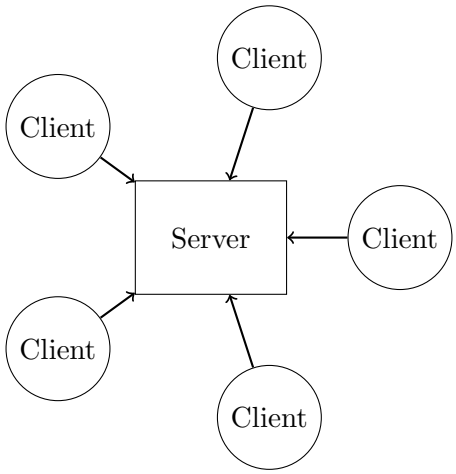


Figure 4.1: A simplified diagram of the client-server architecture.

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 4.2(a)) will be running a single instance of the recommendation engine, and will handle all training and classification events. Multiple clients (Figure 4.2(b)) can then query the classifier running on the server for either hashtag suggestions or

a search query expansion.

Server
- Hashtag recommendation - Search query expansion - Training classifier - Persistent state

Client
- Twitter authentication - Twitter API calls - Send requests to server - Receive data from server

Figure 4.2: 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 that 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 to one day integrate easily 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.

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/search functions in the Twitter API.

Make this diagram neater

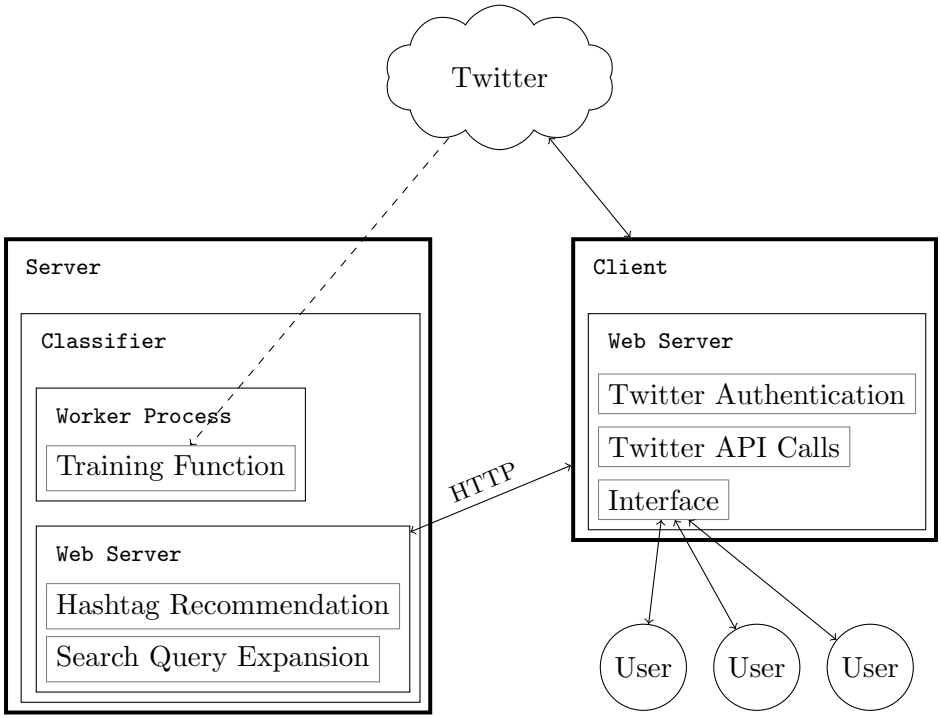


Figure 4.3: An abstract view of the implementation of the system.

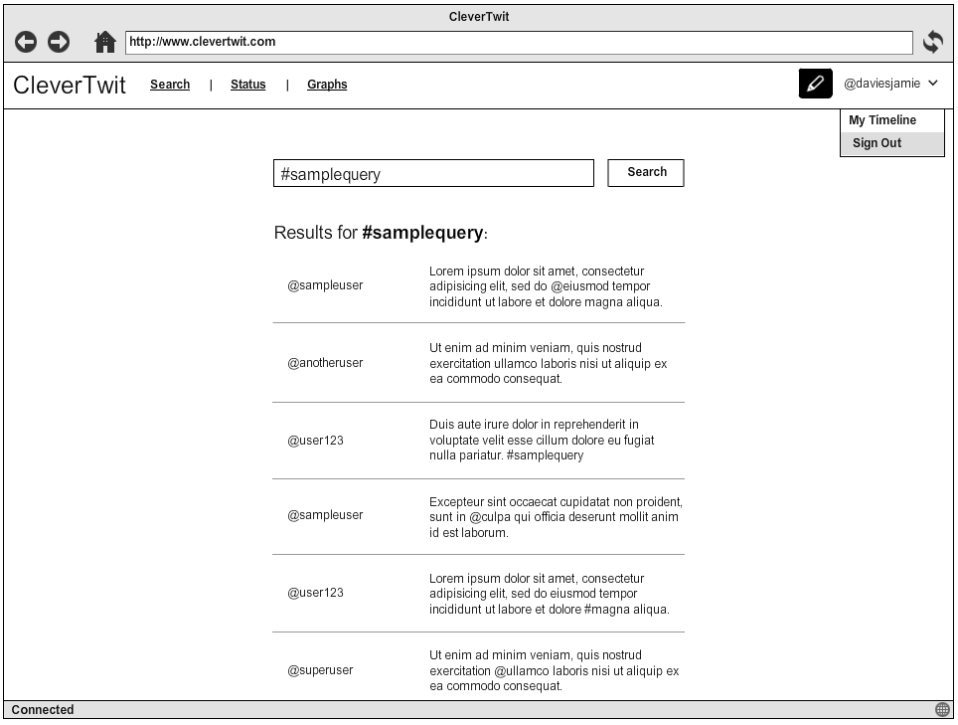
4.3 User Interface

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

Figure 4.4(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 4.4(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 4.4: Wireframe designs of the prototype interface.

Chapter 5

Implementation

This chapter aims to explain how the system was implemented, and the decisions that were made during the implementation process. It will also explain 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 a 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], a thin but functional Python wrapper around the Twitter API, making Twitter API calls easily available within Python. It also provides a

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

set of methods that handle OAuth authentication with Twitter.

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 DOM from within Javascript — it is used specifically within the client to perform AJAX requests to the server, and provide live hashtag suggestions.

Nginx[7] was used to proxy the requests to the client and server, as well as serving all static files. This not only significantly boosted the performance of the system, but it also made secure and accessible.

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

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

by applying a supplied `Function` object to each item in the input stream, to produce another output stream.

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

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 highly customisable, allowing for custom stream implementations to be created — all that is required is for the stream to sequentially output items of data.

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 an infamous problem: 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 object of an arbitrary size that when full, drops any extra items and ignores them. 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, Spout has been made available independently from this project. The full source for Spout is available on GitHub[8].

In addition to open-sourcing Spout, it has also been made available on the Python Package Index (PyPI)[9]. This has been a huge success, with approximately 3000 recorded downloads at the time of writing.

5.3 Classification Server

To choose an appropriate classification algorithm, two main factors were considered. One factor was that the classifier had 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 was that of probabilities: it was 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 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 happened:

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)} \tag{5.1}$$

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:

$$P(C|F_1, \ldots, F_n) = \frac{P(F_1, \ldots, F_n)|C) \cdot P(C)}{P(F_1, \cdots, F_n)} \tag{5.2}$$

Or, in plain English terminology:

[8] `http://github.com/daviesjamie/spout`
[9] `https://pypi.python.org/pypi/spout/0.1.5`

$$P(\text{Hashtag}|\text{Features}) = \frac{P(\text{Features}|\text{Hashtag}) \cdot P(\text{Hashtag})}{P(\text{Features})} \tag{5.3}$$

5.3.1 Feature Extraction

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 other information that is available with each tweet, such as date and time, or the location that the tweet was posted from. However, to keep things (relatively) simple, the feature extraction in this project is based solely upon the actual text that makes up the tweets themselves.

Features are extracted from the tweets by passing their text through the 'twokeniser' system created by O'Connor et al. (2010).[10] This tool matches the text against a collection of complex regular expressions to split the text up into a collection of word-like objects. It is much more complex than splitting on whitespace 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 hashmaps 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 } }`
- 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 } }`

[10] Available online at `http://github.com/brendano/tweetmotif`

It is important to note here that the hashmaps 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 features having to be recalculated every time a single feature count was updated. 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 hashmaps.

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 hashmaps 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:

$$P(\text{Hashtag}|\text{Tokens}) = \frac{P(\text{Tokens}|\text{Hashtag}) \cdot P(\text{Hashtag})}{P(\text{Tokens})} \tag{5.4}$$

$$= \frac{\left(\prod_{i=1}^{N} \frac{\texttt{thc}[\text{Token}_i][\text{Hashtag}]}{\texttt{hc}[\text{Hashtag}]}\right) \cdot \frac{\texttt{hc}[\text{Hashtag}]}{\texttt{hashtag\_total}}}{\left(\prod_{j=1}^{N} \frac{\texttt{tc}[\text{Token}_j]}{\texttt{token\_total}}\right)} \tag{5.5}$$

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

$$P(\text{Hashtag}|\text{Tokens}) = \left(\prod_{i=1}^{N} \frac{\texttt{thc}[\text{Token}_i][\text{Hashtag}]}{\texttt{hc}[\text{Hashtag}]} \right) \cdot \frac{\texttt{hc}[\text{Hashtag}]}{\texttt{hashtag\_total}} \tag{5.6}$$

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 token has only been seen once with a hashtag, then that token will have a 100% probability of being in used in conjunction with that hashtag. This effect was overcome with a weighting function, $P*$, that adds a small bias towards an assumed probability:

$$P*(\text{Hashtag}|\text{Token}) = \frac{(\texttt{weight} \cdot \texttt{ap}) + (\texttt{tc}[\text{Token}] \cdot P(\text{Hashtag}|\text{Token}))}{\texttt{weight} + \texttt{tc}[\text{Token}]} \tag{5.7}$$

Where:

- `weight` is the weighting of the bias, measured in number of tokens.
- `ap` is the assumed probability.
- P(Hashtag|Token) is the standard probability calculated using the method described in Equation 5.6.

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[11], tweets that didn't contain any hashtags, and tweets that weren't 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. For a full list of the stop words used, see Appendix C.

5.3.5 Search Query Expansion

Due to the structure of the internal hashmaps inside the classifier, creating a query expansion is simple. The `htc` hashmap 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 probability and then the top N are returned.

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

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 of the classifier. 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, therefore making the exchange of data between processes difficult.

To successfully share the classifier object between processes, and not have each process operating on a separate object, a proxy object is used. Extending the `BaseManager` class from the multiprocessing[12] 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/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 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 D.

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

5.4 Client

The classification server is designed with an open REST API, 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 E.

The interface itself was built using Twitter Bootstrap to implement the wireframe designs in section 4.3.

Django was used to handle the structure of the pages within the client. In addition, a custom extension of the Django authentication module was created to use the Twitter OAuth API to log users in. This enabled use of the fantastic user library from within Django, but also provided the necessary integration with Twitter. Query expansion and search results are acquired through a Django view that sends a POST request to the classification server, and then a GET request to the Twitter Search API, respectively.

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.

Chapter 6

Testing Strategy & Results

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 a traditional compiled languages, errors in the 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 B.

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 an IPython[2] shell. IPython is a tool 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. These 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, scientifically 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 as thus, 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 it's 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

Project Management

Chapter 8

Evaluation

Evaluation

Chapter 9

Conclusion

Conclusion

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.

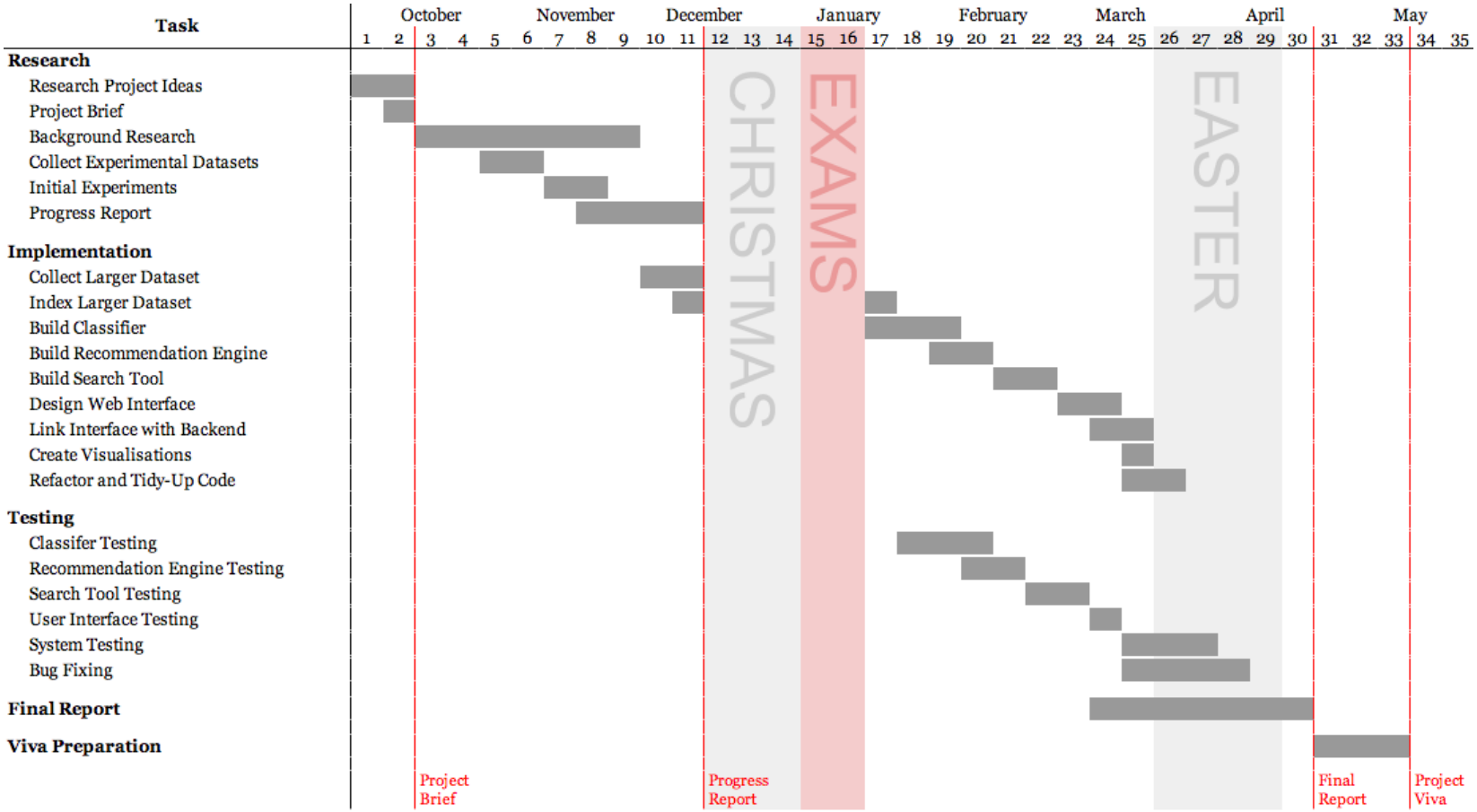
Zangerle, E., Gassler, W. and Specht, G. (2011). 'Recommending #-Tags in Twitter'. *Proceedings of the Workshop on Semantic Adaptive Social Web*. SASWeb '11. Girona, Spain: CEUR Workshop Proceedings, pp. 67–78.

Appendix A

GANTT Charts

Add realistic GANTT chart

This is a Gantt chart showing the scheduled progression through the different aspects of the project.



Appendix B

Tests Results

Add test results

Appendix C

Stop Word List

Add stop words

Appendix D

REST API Interface

Add RESTful API Interface/Endpoints

Appendix E

Client Interface Screenshots

Add screenshots

Appendix F

Project Directory Listing

Add directory tree

Appendix G

Original Project Brief

Part III Individual Project Brief

Enhanced Content Analysis and Information Retrieval Using Twitter Hashtags

Author:
Jamie Davies
jagd1g11@ecs.soton.ac.uk

Supervisor:
Dr. Nicholas Gibbins
nmg@ecs.soton.ac.uk

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 Linguis-