University of Manchester
School of Computer Science
Project Report 2012

Text Mining Twitter for User Perception of Software

Author: Tariq Patel

Supervisor: Dr. Goran Nenadic

Abstract

Text Mining Twitter for
User Perception of Software

Author: Tariq Patel

Twitter is a massive corpus for text and opinion mining.
This project aims to develop a system that text mines Twitter posts to find software or software development tools that have been mentioned by its users and to discover the general sentiment of users towards these software.
This faces many challenges with respect to Natural Language Processing, particularly with the use of informal English.
The development of this system takes a three-stage process from design through to implementation. These stages are Information Retrieval in retrieving tweets from Twitter, Information Extraction in extracting certain features from tweets, before aggregating the data and finally creating a graphical user interface for users to interact with the system.
The features to be extracted include software names, versions, prices and sentiment. Tweets are retrieved from Twitter based upon matching filter terms taken from a dictionary of software and a set of keywords that may be associated with software. This is followed by the analysis of tweets to extract features, before being aggregated and displayed to the user.
The implementation was coded in Java and Python, utilising various public APIs and libraries. The system scores an F-measure of 0.73 for extracting software names from tweets. This measurement was taken from a 10% sampling of the stored data.
There are areas for improvement in the project. Of course the accuracy rating could be improved, but also a more detailed user interface. The system should also aim to extract the purpose behind a user posting a tweet. This would provide more interesting information than the current implementation.

Supervisor: Dr. Goran Nenadic

Acknowledgements

I would like to thank my supervisor, Dr. Goran Nenadic, for his time and guidance throughout the course of this project. Thanks also to his research group for their help. I would also like to thank my friends and family for their continued support and encouragement over the year.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Motivation

The success of a piece of software is based largely upon user opinion. Gathering such information is conventionally done through means of surveying groups of users. However, in the days of social media, people generally express their opinions on popular social networks or microblogging sites such as Facebook and Twitter. This means it is now much easier for companies to receive and collect feedback on products they have developed by monitoring these networks.

Twitter has been at the core of many data mining projects in recent years and this is due to the sheer amount of data produced on a daily basis. Twitter users now post in excess of 340 million tweets every day [Twi12] and as such Twitter provides a massive corpus[1] for opinion mining and sentiment analysis.

1.2 Aim

By text mining Twitter posts for software, users may be able to discover new tools or programs they have not come across before, as well as see reviews by other users.

The aim of this project is to develop a system that text mines Twitter posts to find software or software development tools that have been mentioned by its users and to discover the general sentiment of users towards these software. This system will henceforth be called **SWOT** - Software on Twitter.

1.3 Objectives

In order to successfully complete a project of this magnitude, the task at hand must be split into smaller steps. These objectives are shown in Table 1.1.

1.3.1 Collect and Filter Tweets by Keywords

Collecting tweets is a core task in this project as all the work will be based on tweets stored in a database. Filtering through these is a relatively simple task in that it can be done using

[1] A collection of documents

	Task
1	Collect and filter tweets by keyword
2	Feature extraction
3	Analyse tweet sentiment
4	Structure and integrate data
5	Visualise data through GUI
6	Evaluate the system

Table 1.1: Project objectives

Twitter's APIs, but there are some complexities in ensuring they are all relevant.

The main idea at this stage is to collect tweets based on a set of keywords and software names, programming languages, or company names stored in a dictionary, in order to retrieve relevant, software-related tweets.

1.3.2 Feature Extraction

Feature extracting is the core functionality set out to be achieved in this project. Using rule-based text mining techniques, the aim of this task will be to retrieve up to eight features from every tweet. Tweets will generally not contain all eight features and so the application cannot be expected to fill all of these fields for every tweet. These features will include the software name and version, company name, any programming languages or operating systems, as well as prices, URLs and the sentiment towards the software.

1.3.3 Analyse Tweet Sentiment

Sentiment analysis is another of the more important tasks in this project. This is where tweets are analysed for subjectivity, i.e. whether the tweet is positive, negative or neutral. The **tweet sentiment** feature to be extracted aims to find the general sentiment towards a piece of software, and will be used in the aggregation process in the final stages when trying to establish public perception of the software.

1.3.4 Structure and Integrate Data

The data needs to be structured and aggregated to be able to provide any meaningful output for the user.

1.3.5 Visualise Data Through GUI

Visualising the data is a fairly low priority task in that the system first needs to gather the information. This project centres more around the core back-end development than user experience and as such only a simple user interface is needed in its initial stages.

1.3.6 Evaluate the System

The final evaluation of the produced system will be key in determining the success of this project. The system will be evaluated on the basis of the accuracy of retrieved results, the relevance and novelty of information, and general usability.

1.4 Challenges

There are many challenges facing Natural Language Processing(NLP)-oriented projects.

With the millions of tweets posted every day on Twitter, one can safely presume that many of these will have no relevance to software or any of the other desired information. As such it will be vital to ensure only the most relevant tweets are extracted from Twitter for analysis so as not to waste resources.

Another issue is the world-wide nature of the Internet and microblogging networks like Twitter. This means that several tweets will not be in English and for this reason it would be more difficult to extract features from these tweets. To counter this, it will be necessary to filter tweets not only based on key words but also on their language.

A major issue in NLP research is that of text message shorthand. In a formal document, this problem becomes somewhat irrelevant due to proper usage of Standard English. However, when working with the Twitter platform, the service's 140 character limitation on tweets means users are generally more likely to abbreviate their text and this allows for a lot of ambiguity in the context of each word, and variability in how users may say the same thing. For example, *USA* is both ambiguous and could be represented in various ways. In the case of ambiguity, *USA* could stand for the *United States of America* or the *United States Army*, amongst others. Assuming the former, this could also be represented as *U.S.A.*, thus illustrating the potential problems of both ambiguity and variability.

1.5 Report Structure

This report documents the implementation of a text mining system that is set out to achieve the previously stated goals. The remainder of this report has been split into 6 chapters. Chapter 2 details the general background of this project and previous work in the area. Chapter 3 goes into the design of the software implementation including use case analysis, the architecture of the system and the software engineering methodologies used. Chapter 4 describes the process of implementing each stage of the project and goes into details of how specific aspects such as the Twitter API integration and feature extraction work. Chapter 5 illustrates the results and final outcomes of the project with any meaningful information gained. Chapter 6 provides the general evaluation of the finished project, also outlining the successes and failures of the task at hand. Finally, Chapter 7 concludes the author's conclusions of the project, with suggestions for further work.

Chapter 2

Background

This chapter provides an overview of the text mining field along with previous work in the area and all necessary background information required to understand the major tasks involved in this project. This is followed up with an overview of the different APIs provided by Twitter to work with their platform.

2.1 Text Mining

The information available in the world is growing exponentially, and the majority of this information is unstructured (widely estimated at roughly 80%) [Gri08]. This is where text mining comes in, also referred to as Knowledge Discovery from Text (KDT). "Text mining is the process of extracting interesting information and knowledge from unstructured text" [HNP05]. Its applications tend to work in two steps, first using an Information Retrieval (IR) application to narrow the search space, and then they extract significant parts of the retrieved texts [Pol06]. This general process usually involves structuring a source text by means of parsing and other linguistic analysis, then finding patterns in this structured data and then interpreting this output. An example of the text mining process can be seen in Figure 2.1.

Text mining is fundamentally different from standard web searching in that web searches rely primarily on information that is already known. On the other hand, the goal of text mining is to discover interesting, previously unknown information [GL09]. There is one key issue about text mining; natural language is used by humans for communication and recording information, while computers are incapable of interpreting natural language. Humans are naturally able to find linguistic patterns in text and understand the semantics of what is being said. Computers, on the other hand, face difficulties in interpreting variations in written text through spelling, colloquialism and also the general context of the text. Nonetheless, computers are much more capable of processing large datasets at very high speeds, particularly in comparison to humans. Thus, the objective of text mining is to create an application that is able to apply human-defined rules to large datasets in order to extract linguistic patterns. Machine learning techniques could also be applied, however these are not in the scope of the project as a rule-based approach has been taken.

2.1.1 Information Retrieval

Information Retrieval (IR) is the process of retrieving textual documents which may contain the answers to questions but the process itself does not answer these questions [Hea99]. In-

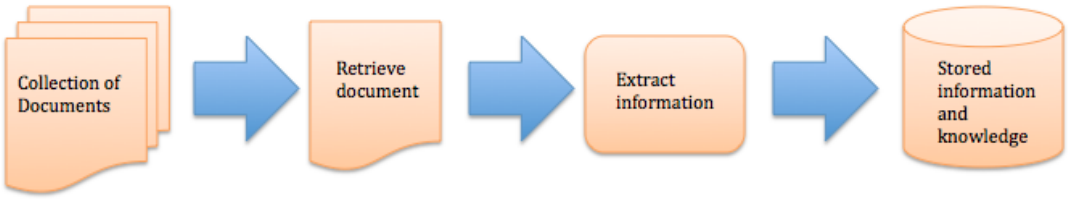

Figure 2.1: The text mining process [GL09]

formation retrieval is fundamentally a web search working off user queries representing an information need. The process works by searching a collection of documents, and then retrieving those matching a user query depending on relevance. The approach to calculating relevance is dependent upon the actual IR engine itself. Generally, this works on the frequency of specific key terms in each of these documents, and usually assigns a relevance rank to each one. This allows for improved results through sorting, particularly when restrictions are placed on the number of results to return.

The IR tasks in this project will mainly be carried out on Twitter's systems, and as such, besides the core concept of IR, its internal specifics are not in the scope of this report.

2.1.2 Natural Language Processing

Natural Language Processing (NLP), in the scope of this project, is the process of extracting information from natural language [WH98], that is, any language written or spoken by humans. This involves parsing and processing unstructured text to be able to gain meaningful knowledge from it. Nowadays natural language processing is done using machine learning techniques. However, in the past, implementations were based on large sets of coded rules. These rules are used to define certain linguistic features in the text in order to understand the semantics behind it. NLP is a major field of research at present and also has applications in both information retrieval and extraction.

There are several public libraries available for NLP. One of these is the Natural Language Toolkit (NLTK), which is a set of Python modules for symbolic and statistical natural language processing [BLK09]. It is an active, well documented project. Alternatives to NLTK include GATE[1] and Minor Third[2]. GATE is equally well documented, however it appears to be a bulky library, and many of its features are not required in the scope of this project. Minor Third on the other hand, is not as well documented, at least at the time of development, and as such would be more difficult to integrate. These alternatives are also Java implementations and as such, may not possess the speed and text manipulation capabilties of Python.

There are many methods involved in NLP tasks and some of these will now be further explored.

[1] `http://gate.ac.uk/`
[2] `http://sourceforge.net/apps/trac/minorthird/wiki`

Tokenisation

Tokenisation is the process of splitting a stream of text into singular words or phrases, otherwise known as tokens. These tokens usually form the basis of further NLP processing. While it can be a straightforward process when using Standard English, the definition of a word, from the tokeniser's point of view, can be somewhat ambiguous. This is particularly true when considering the use of apostrophes. Figure 2.2 shows an example of the different ways of tokenising the word *don't*.

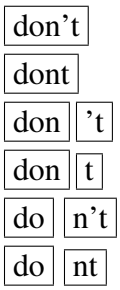

Figure 2.2: The different ways of tokenising the word *don't*

These variations can be problematic in terms of the results being output for certain user queries. For example, in the case of these differing tokenisations of *don't*, a user search for the word *don* would return a positive match twice, but should be negative in its actual context. The importance of normalisation is highlighted when tokenising tweets because a lot of users do not use apostrophes, either due to ease when typing, or in order to reduce the number of characters being used. Thus, varying spellings of the terms should not be tokenised differently.

Normalisation

Once text has been tokenised, these words may need normalising. Normalisation accounts for the several variations in spelling. For example, if you want to search for *Mozilla Firefox* you would want an IR engine to return not only documents containing the exact query but also those containing terms such as *Firefox*, *firefox* and *mozilla firefox*. Failing to normalise terms would obviously yield fewer results, or in the case of information extraction, it may suggest that *Mozilla Firefox* and *mozilla firefox* are two different entities. Thus, normalisation is required to successfully map equivalent classes of terms.

Stop Word Filtering

Stop words are very commonly used words like *a*, *I* and *the* that do not bear any specific meaning on their own. By creating a list of these terms, a *stop list*, a natural language parser can remove these terms from the source text as they hold little or no value in matching queries to documents. In modern systems, however, stop lists are not widely used as they provide little gain in terms of efficiency [MRS08].

Part of Speech Tagging

Part of Speech (POS) tagging is a process typically carried out after tokenisation. Its task is to assign tags to words, for their corresponding grammatical parts of speech, based on both

the word's definition, and its context. This is essentially identifying words as nouns, verbs, adjectives, etc.

However, the process is more complicated than it may first seem. The main issue is that most words do not just have one part of speech, they can have many. For example, *can* could be a noun or a verb, depending on the context it is being used in. Thus, when tagging words, it is important to analyse a whole phrase or sentence. Analysing a word out of context could have significant repercussions. In the context of this project, taking the Microsoft software *Paint* as an example, if someone were to tweet:

> Microsoft Paint has seen some major improvements in its latest release!

This would differ from, say,

> I want to paint something.

where *paint* is being used as a verb. In the first example, seeing that *Paint* is followed by *seen*, a verb, suggests it is unlikely *Paint* is being used as verb. This would be the difference in this project between discovering a piece of software and totally missing it, and thus highlights the importance of context and semantics when analysing text.

The following displays the output of these examples when used with NLTK's POS tagger.

> [('Microsoft', 'NNP'), ('Paint', 'NNP'), ('has', 'VBZ'), ('seen', 'VBN'), ('some', 'DT'), ('major', 'JJ'), ('improvements', 'NNS'), ('in', 'IN'), ('its', 'PRP$'), ('latest', 'JJS'), ('release', 'NN')]

> [('I', 'PRP'), ('want', 'VBP'), ('to', 'TO'), ('paint', 'VB'), ('something', 'NN')]

NNP	Proper noun, singular
VBZ	3rd person singular verb, present
VBN	Verb, past participle
DT	Determiner
JJ	Adjective
NNS	Noun, plural
IN	Preposition

PRP$	Possessive pronoun
JJS	Adjective, superlative
NN	Noun, singular
PRP	Pronoun
VBP	Non-3rd person present singular verb
TO	*to*
VB	Base verb

Stemming and Lemmatisation

Documents contain many different derivations of words, such as *normalise* and *normalisation*, and differing forms of the same word due to its usage or tense, for example, *walked* or *walking*. An information extraction tool should ideally see these as somewhat equivalent terms; this is where stemming and lemmatisation come in. The following example, taken from [MRS08], shows how these techniques should map text:

> am, are, is $\Rightarrow$ be
> car, cars, car's, cars' $\Rightarrow$ car
>
> the boy's cars are different colors $\Rightarrow$
> the boy car be differ color

Stemming is an heuristic process hoping to achieve this goal by simply building basic forms of words by removing affixes like a plural 's' from nouns or the 'ing' from verbs [HNP05]. However, these are not always correct terms. Lemmatisation on the other hand utilises a more sophisticated approach in that it uses a vocabulary and morphological analysis of words, in order to return to the true base form of a word that may be found in a dictionary. This process, however, is much more complex and time-consuming than the former, and stemming may be sufficient for some applications.

N-Grams

N-grams are sequences of n items from a given source text. These items are dependent on the application but are generally letters or words. An n-gram of size 1 is referred to as a *unigram*, while n-grams of size 2 or 3 are *bigrams* and *trigrams* respectively. Larger n-grams are sometimes referred to by the value of *n*, e.g. *four-gram*.

Using the latter of the previous *Paint* examples, the trigram process would yield:

> I want to paint something.
>
> $\Rightarrow$ [('I', 'want', 'to'), ('want', 'to', 'paint'), ('to', 'paint', 'something')]

2.1.3 Information Extraction

The goal of Information Extraction (IE) is to extract specific data from a given corpus. IE can be defined as the task of automatically extracting structured information from unstructured or semi-structured machine-readable documents, generally done through the use of NLP techniques.

In structured texts, information extraction can be fairly straightforward, as labels or tags may delimit strings that need to be extracted [Sod99]. However, in unstructured texts, information is not as clearly understandable by computers and so IE requires techniques from other fields such as machine learning, statistical analysis or those previously discussed from natural language processing.

Typical IE tasks include the following:

Named Entity Recognition

The aim of Named Entity Recognition (NER) is to annotate a source text with markup tags in order to classify strings representing predefined categories such as names, companies, locations, dates and times. For example,

> Cook named new Apple CEO.

would yield the following annotations.

> <ENAMEX TYPE="PERSON">Cook</ENAMEX> named new
> <ENAMEX TYPE="ORGANIZATION">Apple</ENAMEX> CEO.

This example is using the *ENAMEX* tags defined at the Message Understanding Conference (MUC) in the 1990s [GS96]. From the source text it can be seen that *Cook* has been identified as a person and *Apple* as an organisation and structures this text in doing so.

Relationship Extraction

This works with entity extraction in that it works to identify relations between entities. Using the previous example, the relationship extraction process should be able to identify that,

> PERSON named new ORGANISATION CEO.

2.1.4 Evaluation Measures

In text mining, there are several ways of evaluating the accuracy of a system. Before discussing the different measures, few key prediction outcomes must first be defined.

True Positive tp - an entity is predicted to have a certain property, and it in fact does
False Positive fp - an entitiy is predicted to have a certain propery but it does not
True Negative tn - an entity is correctly predicted to not have a certain property
False Negative fn - an entity is incorrectly predicted to not have a certain property
The main measures to be used in the scope of this project are described below.

Precision

Precision is the fraction of retrieved instances that are relevant. This is the number of correct results divided by the total number of results.

$$\text{Precision} = \frac{tp}{tp+fp}$$

Recall

Recall is the fraction of relevant instances that are successfully retrieved. This means it is a measure of the number of correct results that were retrieved out of all of the results that should have been returned. This is the number of correct results returned divided by the total number of all correct results.

$$\text{Recall} = \frac{tp}{tp+fn}$$

Specificity

Specificity is also known as the True Negative Rate. It is the measure aiming to find the accuracy of a system in identifying negative results. This is the number of identified negative results divided by the total number of negative results.

$$\text{Specificity} = \frac{tn}{tn+fp}$$

F-Measure

The F-measure combines precision and recall to find a weighted average of the two measures.

$$F = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

2.2 Sentiment Analysis and Opinion Mining

Sentiment analysis, also known as opinion mining, refers to the NLP application of extracting subjective information in source texts. There are two use cases for sentiment analysis. The first of these is determining whether a text is subjective or objective, that is, whether the statement is factual or opinionated. This scenario is not currently in the scope of the project and leads to the second use case; sentiment analysis also aims to classify the *polarity* of a given text [PL08]. This, to the basic level, involves determining whether an expressed opinion is positive, negative or neutral, but can also be extended to more complex emotions such as happy or sad.

Opinion mining can be done using a weighting system. This method assigns a positive or a negative weighting on a given scale such as -3 to +3. These are applied for each word in the text that relates to the core entity. The text is then given a total score which determines its polarity, and also the strength of the sentiment. In simpler systems the scale may only be from -1 to +1, essentially opting to ignore the strength of the sentiment and just asking for the general sentiment of the text.

There are a number of public APIs and libraries available for sentiment analysis. SWOT will use the Sentiment140 Bulk Classification Service API[3] ahead of others such as AlchemyAPI [Alc11] and the CLiPS Pattern modules. AlchemyAPI results are accurate, however with the massive number of tweets being streamed from the service it is not deemed feasible to continuously make calls to a web service to analyse them for sentiment, as the service only analyses individual tweets. As well as this, there is a limit of 10,000 tweets per day and with the large number of tweets posted on Twitter on the daily basis, this could also be an issue.

While Pattern is an offline system, it states 72% accuracy for movie reviews [CLi12], while the Sentiment140 API is optimised for tweets and boasts 83% accuracy [GBH09]. As well as this, the bulk classification service allows mass analysis with requests consisting of up to 10,000 tweets.

2.3 Twitter Mining

There has been several previous works on text mining Twitter posts, however, the bulk of these have focussed primarily on biomedicine and the financial sector. These works have shown the potential in Twitter has for providing valuable knowledge and information it is felt that software is a new area of interest where Twitter has not previously been used to analyse public opinion. Twitter's low character limit means users have to express their opinions explicitly and encourages the use of emoticons, such as :) or :(, which have been shown to work very well in sentiment analysis [Rea05].

Linguamatics I2E text mining software analyses online media, such as Twitter and blogs, in order to extract reviews, customer comments or praise and any other information [Lin12].

Twitter has also been used to predict the state of the financial stock markets as seen in [**?**]. More examples of text mining Twitter can be seen in [PD11] and [Cul10], where tweets are used to analyse public health and detect outbreaks of the influenza virus.

[3] `http://help.sentiment140.com/api`

2.4 Twitter API

Twitter provides three public APIs for developers to access their massive corpus of data. These are the REST, Search and Streaming APIs, and shall now be further explored.

2.4.1 Search API

The Search API is the simplest tool provided by Twitter. This API is designed to allow users to query for Twitter content and works very much like the search bar found on the Twitter website itself. This content may include a set of tweets with specific keywords or tweets to, from or mentioning a specific user. A simple search would yield up to 1500 of the latest tweets in the last 7 days, which have been cached over a 60 second period. There are, however, restrictions on the rate at which programs can utilise this API [API12].

2.4.2 REST API

The REST API enables programs to access more of the core Twitter functions. This API retrieves not only the information taken from the Search API but also allows building timelines and retrieves more specific user information such as the user's name, profile avatar, tweet count and the number of followers and friends they have. The REST API also allows programs to post on Twitter and carry out other functions like retweeting or favouriting tweets. These extra functions, however, are not required in this project.

2.4.3 Streaming API

Twitter's Streaming API is a real-time sample of all public tweets posted on the sample. It allows filtering in various ways such as user id, keywords or even random sampling, and is regarded as the default option for data mining operations. This is because the Streaming API allows a long-lived HTTP connection unlike the other APIs and as such, programs can constantly remain connected so as to retrieve a running stream of tweets, as the name itself suggests. This removes the overheads associated with reconnecting every time you want to make a query and the API also removes all rate limitations so there is no worry of exceeding your quota. Unlike the other APIs, programs must be authenticated to use the Streaming API.

2.5 Other Technologies

The following sections describe some of the other technologies required in the development of the SWOT system.

2.5.1 JavaScript Object Notation

JavaScript Object Notation (JSON) is a lightweight data-interchange format that is both easy for humans to read and write and for machines to parse and generate [Cro12]. JSON is built on two universal data structures; a collection of name/value pairs and an ordered list of values, equivalent to an array. A basic JSON document may look as follows.

```
{ "foo" : "bar", "id" : 1 }
```

JSON will be widely used in the implementation of the SWOT system due to its usage in the Twitter APIs and compatibility with MongoDB.

2.5.2 MongoDB

MongoDB is a document-oriented NoSQL-type of database, that is, a schema-less design that diverges from the traditional relational database model. MongoDB stores structured data as JSON-style documents with dynamic schemas, thus allowing the integration of data simpler and faster [Ban11].

Chapter 3

Design

This chapter details the overall design of the system to be developed in this project. The software engineering methodology to be used will first be discussed, along with use cases, requirements and the architecture of the system. This will be followed up with notes on the class and database design diagrams. The decisions made with regards to some design choices will also be discussed in more detail.

3.1 Software Engineering Methodology

The software engineering methodology used in developing an application can have many effects on its final outcome. The development of this system will be carried out using principles taken from continuous integration and agile methods such as feature-driven development. There is always a working code repository available for deployment, and all new features to be implemented are to be worked on in clones of said repository. Upon completion of these minor implementations, they are tested to ensure everything is working correctly and assuming there are no issues, the changes are merged into the base repository. This methodology assures developers that if any major issues arise due to recent changes, they will be able to discard all changes and restart if they feel debugging would be a longer process. This ultimately allows for a faster development cycle and provides rigorous testing throughout the implementation process. This development methodology also allows for frequently changing requirements which is to be expected in any development task.

3.2 Use Cases

To serve as a reminder, the aim of this project is to develop a system, SWOT, that text mines Twitter posts to find software or software development tools that have been mentioned by its users and to discover the general sentiment of users towards these software. Table 3.1 revisits the project's objectives, with their respective complexities and priorities.

In order to attain this goal, SWOT must satisfy four uses cases, as seen in Figure 3.1, which have been described below.

	Task	Complexity	Priority
1	Collect and filter tweets by keyword	Simple	High
2	Feature extraction	Complex	High
3	Analyse tweet sentiment	Intermediate	Medium
4	Structure and integrate data	Intermediate	High
5	Visualise data through GUI	Intermediate	Low
6	Evaluate the system	Complex	High

Table 3.1: Complexity and priority of project objectives

3.2.1 UC1 - Stream Twitter

The first use case for the system requires a tool capable of continuously monitoring public tweets and storing these in a database. These tweets should be filtered by language and relevance, that is, tweets should be related to software. These tweets need to be filtered by language to counter any issues faced in the feature extraction stage due to the complexities involved in NLP. This use case is targetted for use by an administrator as the process should be running constantly in the background.

3.2.2 UC2 - Search and Update Software Term

SWOT's second use case requires the ability to search Twitter for tweets concerning user-specified key terms. This is ideally the name of some software that may or may not already be present in the database of extracted software and features. In the case where the software is not already present, it will now be added upon finding tweets. Where the software already exists in the database, the search will still be carried out in order to retrieve more recent tweets matching the search query, in order to update the stored data.

3.2.3 UC3 - Browse Stored Software

The third use case involves allowing users to view the information stored in the database of extracted software and features. This should be displayed in the form of charts displaying the sentiment of tweets and all relevant information found alongside it.

3.2.4 UC4 - View Top Ten Tweeted Software

The final use case for SWOT requires displaying to users the software tools that have been mentioned most often on Twitter. Users should then also be able to carry out use cases 2 and 3 with these tools.

3.3 General Architecture

The system design follows a 3-stage approach, these being tweet retrieval, feature extraction and visualisation for users. These stages are shown in the general archicture model displayed in Figure 3.2. These will be further explored in Sections 3.3.1, 3.3.2 and 3.3.3 respectively.

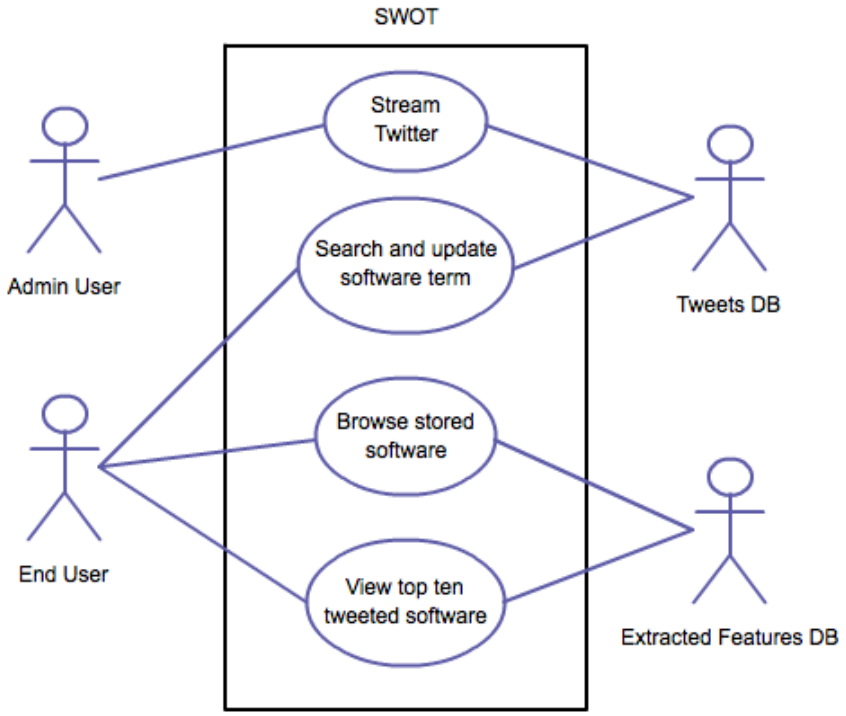

Figure 3.1: Use cases for SWOT

3.3.1 Retrieving Tweets

The first stage involves retrieving tweets from Twitter. The design for this stage follows the same concepts for each of the use cases defined. This can then be split further as seen in Figure 3.3. The program should retrieve a set of search terms from a dictionary along with some keywords that may be associated with software. This dictionary consists of a list of software, companies, operating systems and programming languages. The full list of keywords and dictionary items can be seen in Table 3.2 and Appendix A respectively. The respective counts for each of these types of items is shown in Table 3.3. These are to be used to form a request for tweets from Twitter. Twitter will respond with the corresponding tweets and data, which are to be checked for language, to ensure they are in English. The remaining set of English tweets are then to be further parsed to extract the required information for storing these tweets in the database.

alpha	api	app
beta	game	mac
pc	release	sdk
software	source code	version

Table 3.2: List of keywords

This returned information will be stored in a relational database and its design is shown below in Figure 3.4. Tweets will not be stored alone but also with simple user information

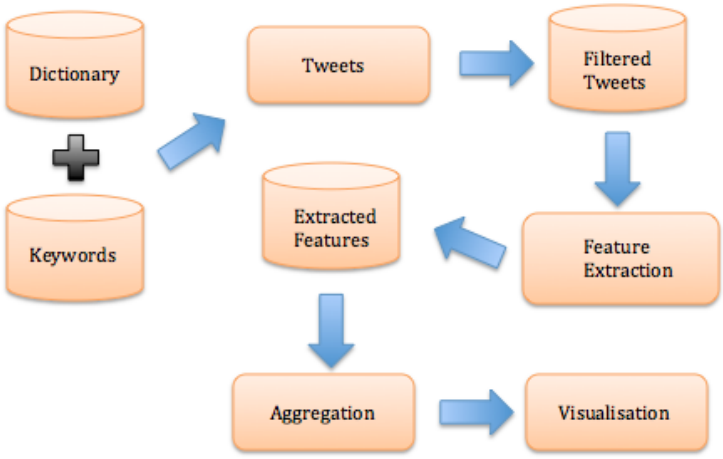

Figure 3.2: General architecture of the SWOT system

Type	Count
Software	389
Operating System	12
Company	25
Programming Language	448
Keywords	12
Total	**886**

Table 3.3: Number of terms for dictionary types

to allow for future user profiling for a more targetted approach to tweet retrieval. The actual tweet information to be stored is its id on the Twitter platform - allowing for cases where users delete their tweets - its text content, time of creation, user id and the keyword used to find it, where one was used. There will also be fields for sentiment, i.e. positive, negative or neutral, and sentiment strength, where weightings have been used. These fields should default to `NULL` as their values will be computed at a later time. Finally, there is a *tagged* boolean flag which signifies the given tweet has been processed, and the feature extraction process has been carried out on it. This is to work in conjunction with the *found* boolean flag which indicates whether or not software has been found in the tweet. This is to be implemented in MySQL due to its simplicity, compatibility with the design and also because it is readily available on the university computers where data can be easily accessed both locally and remotely.

3.3.2 Feature Extraction

The feature extraction stage will execute the task of extracting information from tweets. The aim at this stage is to extract the following features:

- **Software name** - This discovery is the main purpose of the project and as such is vital.
- **Software version** - The version of the software is important because major changes may have been made over the course of a few releases, and so it is necessary to note which

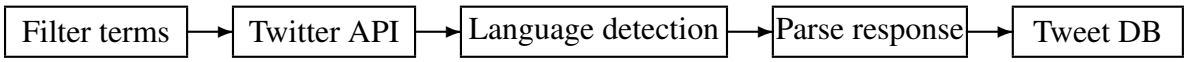

Figure 3.3: Design for tweet retrieval

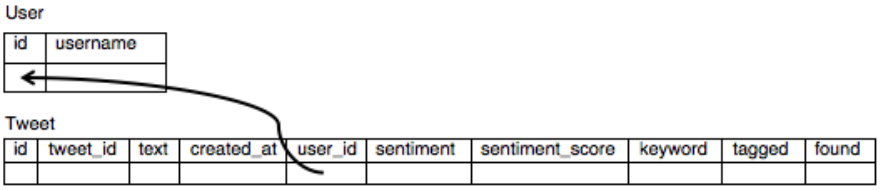

Figure 3.4: Database design for storing tweets

release people are referring to.

- **Company name** - This is not a major feature, however it may be interesting to know who developed a certain piece of software. It may also be used where a user of the system wishes to find public sentiment towards a company as opposed to some specific software.
- **Programming language** - This feature ideally signifies the language or languages in which the found software was developed. However, as with the company field, this may be used to find sentiment towards specific programming languages or practices.
- **Operating system** - Operating system (OS) extraction works similarly to company and programming language extraction in that its expected use is to find the operating systems upon which the found software runs, but it can also be used to find the sentiment towards a specific OS.
- **Price** - This is geared towards retrieving information about the user.
- **Relevant URLs** - URLs are also extracted for extra information.
- **Tweet sentiment** - This is another important feature in terms of the overall project goals. The tweet sentiment aims to find the general sentiment towards a piece of software in a tweet. This will also be used in the final aggregation stages when trying to establish public perception of the software.

Extracting these features will involve the stages described in 2.1 and the general design of this process can be seen in Figure 3.5.

Tweets will be analysed for sentiment after they have been retrieved from the tweet database. This is to be done using the Sentiment140 API, as previously discussed in Section 2.2. As this is a bulk classification service, analysis is to be done on all the tweets retrieved at once, as opposed to individual tweets. This may seem inefficient in the sense that a tweet may be analysed even if it does not contain any references to software. However, using the bulk classification service allows the total number of web requests to be greatly reduced, thus also reducing the overhead when making each request. This will ultimately improve the efficiency of the system.

After this step, URLs need to be extracted from each tweet. URLs provided in tweets related to software are likely to link to further reviews or the application's download page.

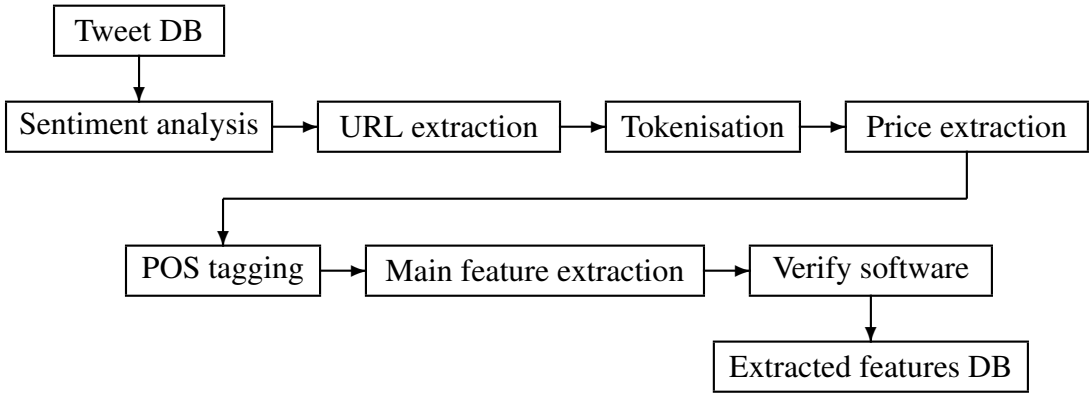

Figure 3.5: Design for feature extraction

These should then be removed from the text before further processing as it may cause conflicts when extracting other features.

The text should then be tokenised in order to produce a list of tokens upon which the remaining processing will take place. Prices are the next feature to be extracted. The tokenisation process itself should be able to isolate prices of the form *£10* or *$0.99* and so this stage will not require much extra processing of the text.

After extracting these two features, the tokens are to be tagged for their part of speech. This will allow for improved entity recognition as linguistic patterns may be applied to the extraction process. The main feature extraction stage has the task of extracting all of the remaining features detailed above. This will require n-gram modelling and linguistic rules and patterns to be able to find candidates for software and other key entities in the text.

These candidates must be verified. If they belong in the dictionary, this is sufficient. However, in cases where the candidate is a new item, other means of verification are required. To do this, SWOT will use the Microsoft Bing API to query the web for these candidates. If the results suggest said candidate is in fact a piece of software, the system will tag these names as software and store the extracted features. In other cases, these features will be discarded.

Due to the volatile nature of the information being extracted from these tweets, this data should be stored in a MongoDB database. As a result, there is no formal design for this database structure, however, it is required of the system to at this stage store any features it has managed to extract along with the key information associated with the tweet that had been stored in the tweet retrieval stage, such as its unique id and text content.

3.3.3 Aggregation and Visualisation

The visualisation stage has the task of displaying all the gained information and knowledge to the user. Ultimately, it must also provide a graphical interface for users to interact with the system in order to perform their own searches.

There are two parts to this phase of project. Firstly, the data gathered must be *aggregated* so that meaningful knowledge can ultimately be represented. Upon completing this task, a *graphical user interface* must be designed for users to interact with the system, perform their own queries, and look at what information has been found.

Aggregation

Aggregating the results is the process of bringing together all of the different data sources for data on a single output entity such as a piece of software. This aggregated data can then be used easily by the GUI to display information to the user. In this project, this task will entail grouping the stored documents by software or other similar information such as company or operating system. Once this information has been retrieved, statistics and charts expressing sentiment can be derived. The main task here is to create pie charts for each piece of software to show the sentiment of the tweets mentioning them. Other details include the most tweeted pieces of software.

Graphical User Interface

Upon the completion of the aggregation tasks, these charts and statistics need to be passed to a Graphical User Interface (GUI). The GUI of choice for this system is a web application, as opposed to a desktop application. This is a more extensible solution as changes would not need to be pushed out to all users.

There will be three actions for users to carry out on this GUI. These will relate to use case 2 of this system, as seen in Section 3.2.2. The first action to carry out will be searching for tweets posted on Twitter. A mockup of this action can be seen in Figure 3.7. Once these tweets have been retrieved from the Twitter Search API, they will be displayed on the web page, and in turn features will be extracted, and again displayed to the user.

After this, there will be an analysis page. This page will show a list of software or operating systems, from which a user can select to view charts displaying sentiment and a list of tweets. These charts will have been creating in the aggregation stage, but should be dynamically created. This page is shown in Figure 3.8.

Finally, once a number of tools, that is, software, operating systems etc. have been found in tweets, the home page should display a list of the most tweeted of these items. Clicking on an item in this list should offer the option of searching again for it, in order to update information, or to view its charts and sentiment information. This should look as the wireframe in Figure 3.6.

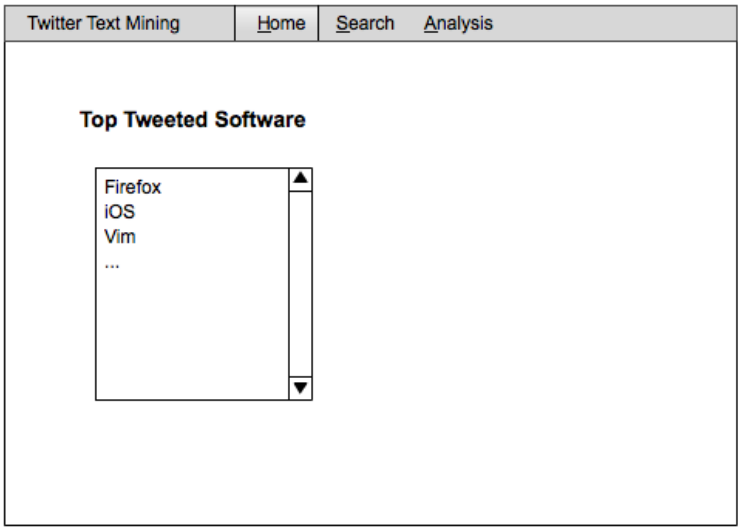

Figure 3.6: A mockup of the website's home page

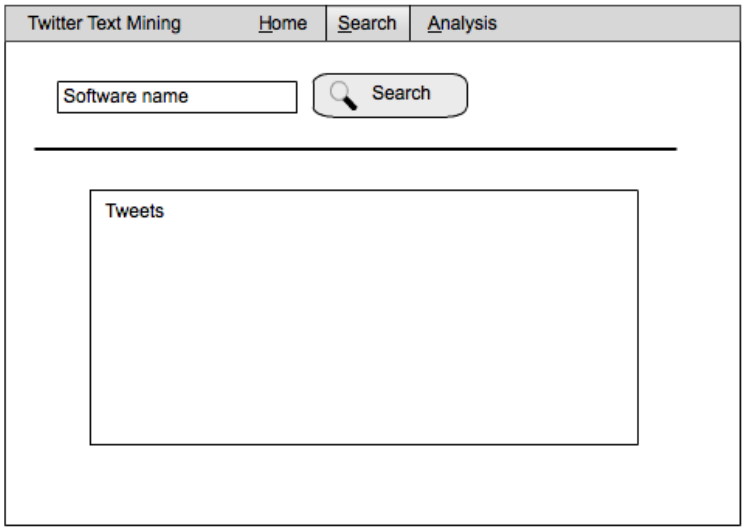

Figure 3.7: A mockup of the website's search page

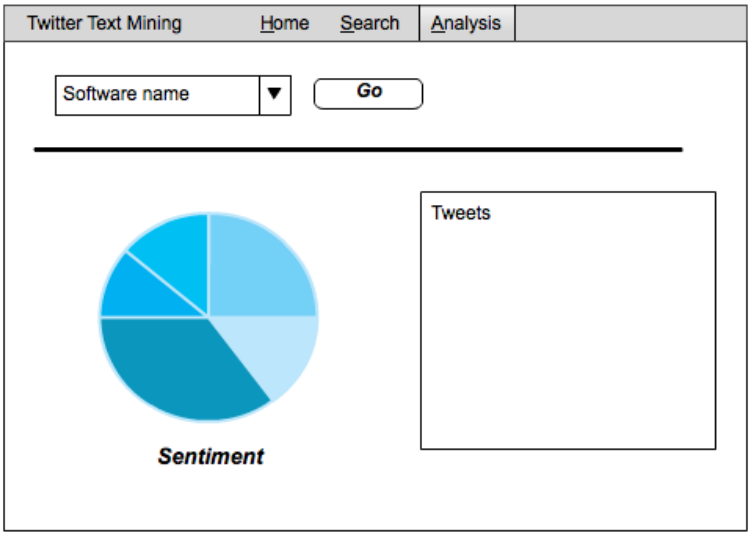

Figure 3.8: A mockup of the website's analysis page

Chapter 4

Implementation

This chapter outlines the main stages in implementating the designed system in code.

The system to be developed, as initially described in Figure 3.2, is fairly representative of a Question-Answering (QA) system. QAs are data mining systems which use Information Retrieval (IR) and Information Extraction (IE) techniques to answer user queries. As such, this project was implemented in 3 stages, each corresponding to these subsystems in QA systems. The first of these was retrieving tweets, the IR phase of this project. Upon successful retrieval, information, the required features in this case, must be extracted and finally, these results must be displayed to the user in a simple, straightforward fashion. These stages will be further explored as follows:

4.1 Tweet Retrieval

Without tweets, there is no work to be done, and so retrieving tweets can be regarded as the most important part of this project. The main objective of this stage is to retrieve as many relevant tweets from Twitter as possible. To do this, the system will interact with the set of public APIs Twitter provides in order to fulfill the requirements of each use case stated in Section 3.2. The system is designed to use all of these to fulfill the requirements of each use case. This subsystem in the project is implemented in Java. This is because of its strong object oriented nature and platform independency. Of course, there are other options such as Python, however Java is a programming language with relatively straightforward multithreading capabilities.

4.1.1 Streaming Twitter

The Streaming API allows the system to fulfill the requirements of having a fully automated system that constantly monitors Twitter for software-related posts, as described by use case 1, in Section 3.2.1.

The implementation at this stage utilises Twitter's filtering URL at `https://stream.twitter.com/1/statuses/filter.json` and passes it the set of dictionary terms and keywords to filter tweets by described in Section 3.3.2.

This implementation could have been done using the *Twitter4J*[1] Java library for Twitter integration, however most of the functions appear unnecessary and excessive in the scope of

[1] `http://twitter4j.org/`

this project. For this reason, the Twitter Streaming API integration was implemented from the ground up.

Upon retrieving these filter terms from the database, the application formats this list into a string after which it creates three new Thread objects, a *DatabaseThread* which will carry out all database operations, a *StreamParseThread* which parses the stream of responses sent back from the Twitter server, and a *ScannerThread*, which monitors the running state of the program, so as to allow a clean exit when the user wishes to quit. This scanner thread simply monitors the console input for users to type the exit command, upon which all connections are dropped and final parsing and database operations are carried out before closing the application. This high level control flow can be seen in Figure 4.1.

On initialising these threads, the application attempts to set up a secure connection to Twitter using the HTTPS protocol. It uses the POST method to write the string of filter terms to the server in order to being receiving tweets. Once this connection is fully set up, Twitter will return JavaScript Object Notation (JSON) strings for each tweet, and so a JSON parser is set up using the Google-Gson Java library [SLW11]. The aforementioned threads are then started as the actual streaming process now begins.

For every tweet returned by the API, the application adds this JSON response, as a *JsonObject*, to a queue in the *StreamParseThread* class, using the following simple method:

```
private final List<JsonObject> parseList = new ArrayList<JsonObject>();

public boolean addTask(final JsonObject object) {
    synchronized (parseList) {
        return parseList.add(object);
    } // synchronized
} // addTask(JsonObject)
```

Listing 4.1: Adding tweets to a parse queue

The parser thread now assumes control of the processing to be done, while the main thread continues to add to this *parseList* queue. The parser thread has the sole task of parsing the information in this JSON object into a more meaningful *Tweet* object. This class' properties can be seen in Listing 4.2. To do this, the JSON object first needs to be checked if it represents a tweet delete entity, that is, a object containing the "delete" key signifying a user has deleted their tweet. In such a case, Twitter requests that applications honour the user's requests and delete this tweet. If otherwise the JSON object is actually a tweet, the program extracts the Twitter user's details to check their locale. In cases where this is not English, a `null` value is returned and the tweet is ignored. If this test passes, all the remaining properties described in the *Tweet* and *User* classes are extracted and returned as a single Tweet object.

This Tweet object is encapsulated in an *InsertKeywordTask* object. This class is an implementation of the *DatabaseTask* interface, which is used to perform the different database operations when used in conjunction with the different *DatabaseConnector* classes. The hierarchy of these classes can be seen in Figures 4.2 and 4.3 respectively. To further clarify, the *DatabaseThread* constructor takes a *DatabaseConnector* object as an argument. This allows for a more extensible system as different types of database management systems can be used with the application. It must be noted that in the current implementation, the *DatabaseThread* class has been implemented at a similarly high level of abstraction, and as such an *SQLThread* extends this class for use with a MySQL database. The *SQLThread* initialises with a *TweetSQLConnector* object, as it only operates in classes related to tweet retrieval. The DatabaseThread class maintains its own queue of *DatabaseTask* objects as the parser thread, previously seen in

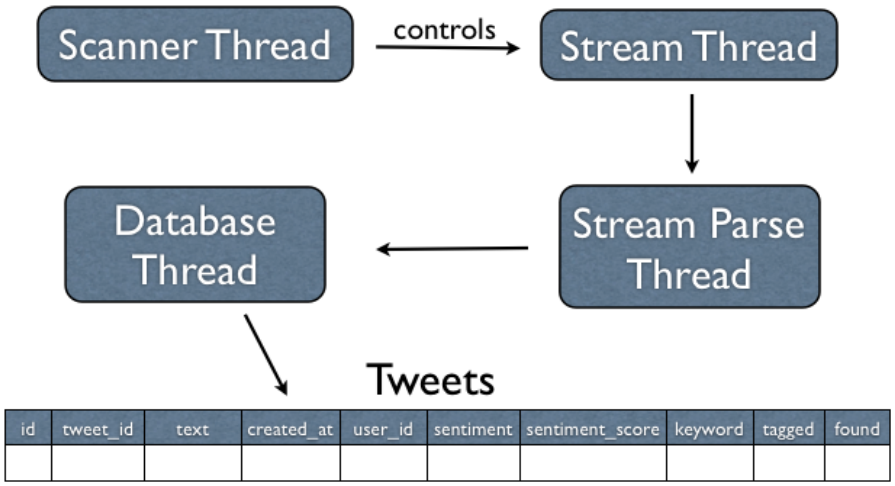

Figure 4.1: Control flow for tweet retrieval subsystem

Listing 4.1.

```
public class Tweet {
    private final long tweetId;
    private final String tweet;
    private final String createdAt;
    private final User user;
    private String keyword = null; // Only used when filtering
    ...
} // class Tweet

public final class User {
    private final long id;
    private final String username;
    ...
} // class User
```

Listing 4.2: Tweet and User class properties

Class design aside, once this *InsertKeywordTask* object has been created, it is added to the queue in the *DatabaseThread*. The respective implementations of the *doTask()* method in each of the DatabaseTask classes will be performed as this queue is emptied. In the case of InsertKeywordTask, this is simply `db.insertTweet(t)`, where `db` is the TweetDatabaseConnector passed to it in the `doTask()` method, and `t` is the Tweet object it was initialised with.

This process is completed when the *TweetDatabaseConnector* inserts the tweet into the database, in the current implementation using Java Database Connectivity (JDBC) to manipulate the MySQL server.

4.1.2 Searching Twitter

The Search API will now be used to handle use case 2 of the designed system, as described in Section 3.2.2. With the Search API not operating in real time, the realisation of this use case can afford to be a much simpler system. The levels of multithreading displayed in streaming

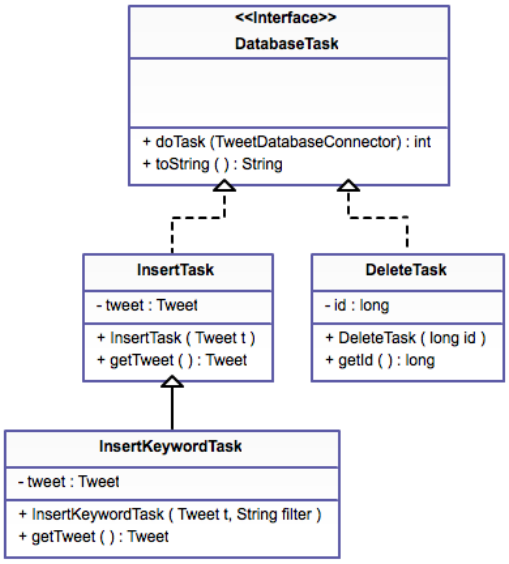

Figure 4.2: DatabaseTask class diagram

Twitter will not be required, as interaction with the API is more of a serial communication as can be seen in Figure 4.4.

The implementation of the Twitter search use case utilises the Twitter Search API URL at `http://search.twitter.com/search.json`. The API also offers eXtensible Markup Language (XML) format responses, however, working with JSON allows consistency within the system. As with the implementation of the Twitter stream use case, the application begins with the initialisation of a *DatabaseThread* and is initially given a single keyword which will be searched for. This keyword is chosen by the end user. A simple HTTP GET request is then made to the above URL and Twitter then returns up to 1500 tweets from the last seven days corresponding to the search term.

The Search API returns a different JSON response to that of the Streaming API. Each JSON string contains an `iso_language_code`, and this will be used to filter tweets by language. Once the desired information has been extracted, i.e. the properties of the *Tweet* class, the remaining operations are carried out just as they are in the realisation of the Twitter streaming use case, that is, the tweet is encapsulated in an *InsertKeywordTask* object and this is added to a queue in the *DatabaseThread* for the insert task to be carried out.

4.2 Feature Extraction

Once tweets have been retrieved from Twitter and stored in the MySQL database, they are now available for feature extaction, which can be regarded as the core stage in implementing the system. This subsystem involves using NLP techniques to extract the information described in Section 3.3.2 from each tweet. This subsystem is implemented in Python 2.7 due to the raw power it possesses and also due to the decision to use the Natural Language Toolkit (NLTK). It was felt that Python's speed and text manipulation would allow for a better implementation of the system.

There are many steps involved in implementing the feature extraction. These are explored

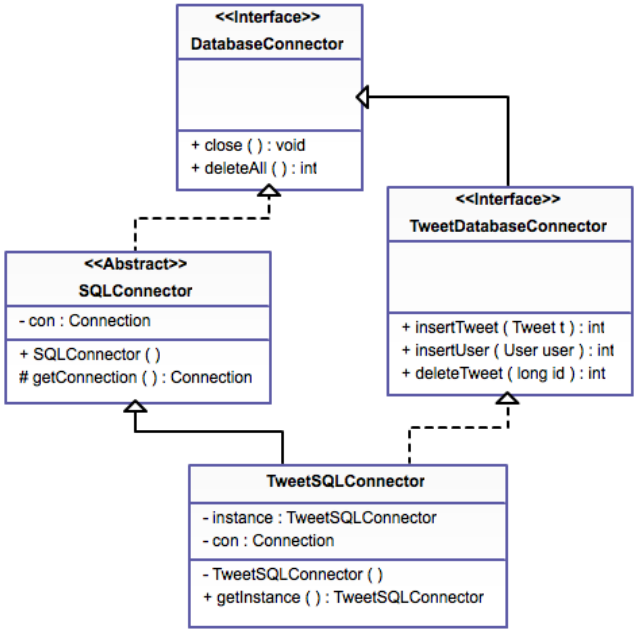

Figure 4.3: DatabaseConnector class diagram

in order of execution.

4.2.1 Sentiment Analysis

Sentiment analysis was recognised as one of the key features to be extracted from the initial design stages (see Figure 3.5). It has been implemented using the Sentiment140 Bulk Classification Service API. The sentiment analysis of tweets is carried out before any of the other feature extraction work. As tweets have been retrieved and stored in the MySQL database, this part of the system selects the latest tweets, retrieving just the id and text, that have yet to be analysed and packs them into a JSON string object of the format:

```
{ "data": [ { "id": "1234", "text": "Google Chrome is awesome!" },
            { "id": "1235", "text": "Safari 5.0.2 is out now" },
            { "id": "1236", "text": "I really hate the new Firefox" } ] }
```

This JSON string is then posted to the Twitter Sentiment API where classifications into the positive, negative and neutral classes are carried out by a Maximum Entropy classifier, using unigram and bigram features, trained with tweets containing emoticons. The internal specifics of a Maximum Entropy classifier, however, is not in the scope of this project.

Currently only 100 tweets are analysed at a time due to time constraints when users wish to run the program in real time. By using a small number, the data needing to be transferred is minimal and allows for a more interactive user experience.

Using the previous example, the data is returned by the server in the following format, with a polarity field added to each analysed tweet with values 0, 2 and 4 corresponding to negative, neutral and positive respectively.

```
{ "data": [
```

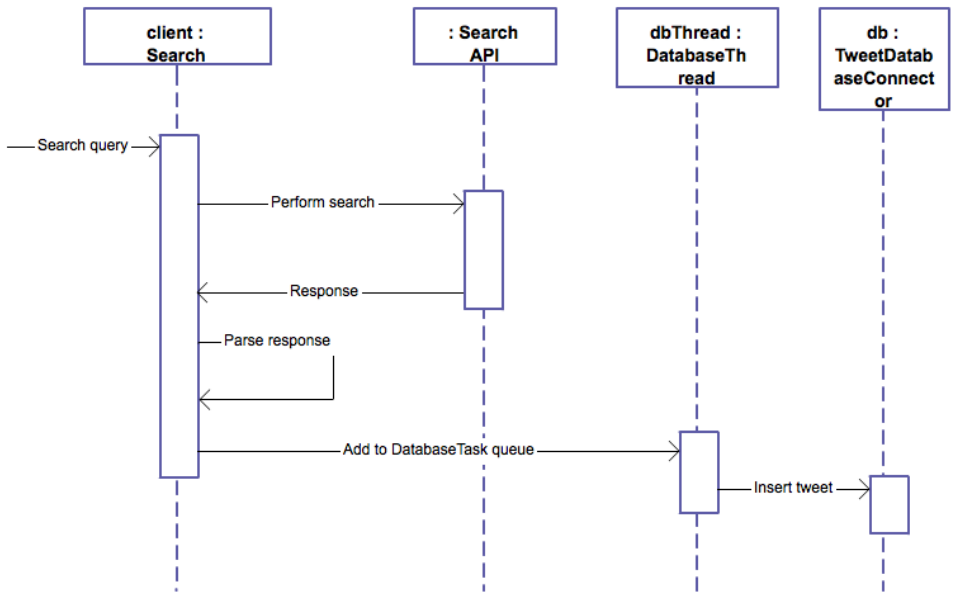

Figure 4.4: Searching Twitter sequence diagram

```
  { "id": "1234", "text": "Google Chrome is awesome!", "polarity": 4},
  { "id": "1235", "text": "Safari 5.0.2 is out now", "polarity": 2 },
  { "id": "1236", "text": "I really hate the new Firefox", "polarity": 0 }
] }
```

Upon receipt of this response, the JSON formatted string is parsed and the corresponding record for the tweet previously stored in the MySQL database is updated with new values for sentiment score and the actual sentiment, using polarity and its semantic meaning respectively.

4.2.2 URL Extraction

Before extracting context and semantics from tweets, any URLs mentioned are found and removed. Assuming the tweet is software-related, these URLs are quite likely to be links to the software, or further reviews. This task is done using NLTK's `regexp_tokenize()` function with `http://[^ ]+` passed as the regular expression that finds URLs. If the tweet is later found not to contain any software, these URLs are discarded. Potential issues with this implementation could be that a user may post a URL without the preceding `http://` protocol prefix and these would not be found by this regular expression. However, Twitter automatically converts URLs to their `http://t.co/` domain and so this is resolved on the Twitter server side.

4.2.3 Tokenisation

After URLs have been extracted and removed from the source text, the tweet is tokenised to produce an array of all the terms in the tweet. The tokenisation process is also done using NLTK's `regexp_tokenize()` function, passing it the regular expression - `\w+([.,]\w+)*|\S+`. This expression returned superior results to alternation tokenisation functions provided by NLTK, such as `wordpunct_tokenize()` as it was capable of finding numbers and currencies without splitting them. Using the above example,

I really hate the new Firefox

this would be tokenised to the following:

['I', 'really', 'hate', 'the', 'new', 'Firefox']

The following example shows a more complicated tokenisation process.

Norton Anti-Virus released for $50 #ripoff

⇒ ['Norton', 'Anti', '-Virus', 'released', 'for', '$50', '#ripoff']

4.2.4 Price Extraction

Continuing on from this tokenisation of the original source text, the current subsystem attempts to find prices in the array of terms. This is done using Python's built-in regular expression module, `re`. A number of regular expressions are used to define patterns denoting numbers, currencies and quantifiers like 'hundred' and 'thousand'. As the form of prices vary, for example in the case of mobile apps you might find '£0.59', '59p' or even '59 pence', these combinations of tokens may be split across two tokens in the array returned from the tokenisation process. For this reason, it is necessary to iterate over all items in the list of tokens while remembering the previous one. This obviously means a less efficient system, however, it has produced the best results in such variable conditions.

4.2.5 Part-of-speech (POS) Tagging

The POS tagger used by this system is taken from the NLTK modules and uses the `pos_tag()` function which takes a tokenised sentence as its only argument. Continuing from the first example, this process tags as follows:

['I', 'really', 'hate', 'the', 'new', 'Firefox']

⇒ [('I', 'PRP'), ('really', 'RB'), ('hate', 'JJ'), ('the', 'DT'), ('new', 'JJ'), ('Firefox', 'NNP')]

PRP	Pronoun
RB	Adverb
JJ	Adjective
DT	Determiner
NNP	Proper Noun

4.2.6 N-Grams

The implementation of creating n-grams in this project is done using the `nltk.util.ngrams()` function. This process starts by creating a five-gram of the tweet tokens. This means a sequence of five tokens will be created from the array of tokens. The system utilises a five-gram sequence due to potentially long software names, basing this on the naïve assumption that these names will not exceed five words. This will allow for improved extraction of software names in the next stage. Using the Firefox tweet as a running example, the outcome of this five-gram modelling process can be seen below.

[‘I’, ‘really’, ‘hate’, ‘the’, ‘new’, ‘Firefox’]

⇒

[((‘I’, ‘PRP’), (‘really’, ‘RB’), (‘hate’, ‘JJ’), (‘the’, ‘DT’), (‘new’, ‘JJ’)),

((‘really’, ‘RB’), (‘hate’, ‘JJ’), (‘the’, ‘DT’), (‘new’, ‘JJ’), (‘Firefox’, ‘NNP’))]

4.2.7 Main Feature Extraction

This tagging process consists of the core functions of the proposed system. Its purpose is to extract all the features that have yet to be extracted, that is, software names and versions, companies, programming languages and operating systems. It also attempts to find any prices that may previously have been missed, and also has the task of discovering software that is not already in the dictionary.

Having created a set of five-gram sequences from the tweet, the application may now iterate through each of these in an attempt to find any information that has not yet been found. For each of these sequences, the program iterates through each POS-tagged token in the sequence. The tagging process then proceeds as follows:

if token is tagged as a noun **then**
  if token is in dictionary of software, companies, os, programming languages **then**
    if previous token not tagged as determiner or preposition **then**
      Feature has been found
    end if
  end if
end if

This rule filters out linguistics of the form shown in Figure 4.5. If however, these conditions

Figure 4.5: A linguistic filter

fail, usually in the case where none of the tokens are in the dictionary of keywords used to retrieve these tweets, a regular expression is used to find clues to the presence of new software.

```
^download$|^get$
```

The above regular expression matches on the words *download* or *get*. This works on the basis that many tweets about software are usually posted to promote said software. This is generally done by urging others to download it, and that too by means of application stores like the App Store, or Google Play. This then allows the next token to be analysed to check if it is in fact a piece of software. This is done by checking that token’s part of speech tag, and if it is a noun, the following tokens are also assessed in case the name of the software is longer than one word. This possible software name is then noted and kept aside for verification by web search as discussed in Section 4.2.8. This regular expression can be applied to the five-gram in conjunction with others in order to maximise the number of features extracted. The following expression could be used to find an operating system.

```
^on$|^for$
```

Figure 4.6: A linguistic pattern to find software and the operating system it may run on

By applying these together in the form displayed in Figure 4.6, the system may be able to determine the platform upon which a piece of software runs.

Once software has been found in the tweet, a search for its version number begins. This essentially works on the assumption that once software has been named, if a version number is to appear, it will be stated either immediately afterwards, or after the word *version*, or some derivative, such as *ver* or *v*.

Finally, this section of the implementation does one more check for prices, this is mainly for free software, as the word *free* would not be found at the price extraction stage explored in Section 4.2.4.

```
^is$|^for$
^free$
```

The above regular expressions are used similarly to the usage of the expression finding operating systems that software may run on. If the first expression is matched in the text, it is likely the next token is a price. In cases where the next token is *free*, it will match the second regular expression, thus signifying the price of the software. This is a naïve approach but in most cases the prices will already have been extracted.

This concludes the main feature extraction process leaving just the verification of new software names via a web search. Listings 4.3 and 4.4 show examples of the data structure storing these extracted features.

```
{
  "company_id" : "23",
  "company_name" : "apple",
  "os_id" : "9",
  "os_name" : "ios",
  "software_name" : "ios",
  "tweet" : "There are some new wallpaper in Apple iOS 5.1. Check them out!
     ",
  "tweet_db_id" : "439640",
  "version" : "5.1",
  "sentiment": "positive"
}
```

Listing 4.3: Example of some extracted features

```
{
  "price" : "free",
  "software_id" : "159",
  "software_name" : "moodle",
  "tweet" : "Training hundreds or thousands of staff?? If you want easy,
     affordable e-learning training then download moodle free http://t.co/
     beZgqaOd",
  "tweet_db_id" : "440065",
  "url" : "http://t.co/beZgqaOd"
}
```

Listing 4.4: Another example of some extracted features

4.2.8 Software Verification

The feature extraction subsystem may discover new software, and as such needs to verify these are actually pieces of software and not something else. To do this the program utilises the Microsoft Bing API which returns web search queries. As the main tagging process checks the dictionary for matching software names, and the tweet retrieval engine uses both the dictionary and a set of keywords, there will be some pieces of software mentioned in the tweets that are not in the dictionary. As a result, these will be flagged as possible software names, and then queried on the Microsoft Bing search engine with the keywords "movie", "music", and "software game". These keywords were selected on the basis that the initial search key terms retrieved many tweets referring to music and films. The implementation of the Bing API integration can be seen in Appendix B.

```
function BING_SEARCH(bing, term)
    music = bing.search(term, music)
    movie = bing.search(term, movie)
    software = bing.search(term, software game)

    if size(software) greater than size(movie) and size(music) then
        if references to software in title and description then
            return True
        end if
    end if

    return False
end function
```

If the number of results for software associated with the searched term is greater than corresponding results for films and music, the results are checked for identifiers of software in their headings. Therefore if any of the results suggests the searched term is a piece of software, that is assumed true.

4.3 Storing the Extracted Information

As the information being extracted is temporarily stored in a Python dictionary variable, it is essentially in the form of a JSON string. The database design for storing this information is also in the form of a NoSQL database. For this reason, a document-based database system seems to be the best approach. By using MongoDB, it is easy to store the extracted information, as it is as straightforward as directly storing the string representation of this variable as a record in the database. Once this information has been successfully stored in MongoDB, the boolean *tagged* flag is set to True for the corresponding tweet in the MySQL database.

4.4 Visualisation/Graphical User Interface(GUI)

The final stage of the project is to aggregate and present the results to the user in a GUI.

4.4.1 Aggregation

The aggregation process has been implemented in Python because the MongoDB wrapper class has already been written and due to the simple mapping between Python dict types and MongoDB's stored documents.

This stage involes two processes. The first of these is finding the general sentiment towards a particular piece of software from all of the tweets mentioning said software, as in use case 3 (Section 3.2.3). This is a simple *find* query and the command and syntax for carrying out this task in Python is equivalent to its MongoDB counterpart. The following command would retrieve documents containing the key-value pair of `software_name=firefox`, given the collection, MongoDB's equivalent of a table in a relational database, is named `tagged_tweets`.

```
db.tagged_tweets.find( { 'software_name': 'firefox' } )
```

Upon retrieving these documents, the application extracts the distinct values from the `sentiment` field of each document, which can be *positive*, *negative* or *neutral*. For each of these values that appears, its appearance count is calculated using a looping function and these counts are used to make charts in the web application, the topic of discussion in Section 4.4.2.

The second task, use case 4 (Section 3.2.4), is to find the top tweeted software and also includes operating systems in the final implementation. This is completed using a grouping function defined in the Python MongoDB driver module.

```
def _group(self, key):
        return self.tags.group(key=[key],
                               condition={key:{'\$ne':None}},
                               initial={'count':0},
                               reduce='function(obj,prev) { prev.
                                   count += 1;}')
```

The above function returns a list of items grouped by a given `key`. The items in the list are also given a `count` value which is the number of times that software name will have appeared in the database collection. This list is then sorted in descending order and a slice of the top ten of these results are returned to be displayed to the user.

4.4.2 Web Application

The web application is required to work with both the Python and the Java code in this project. As such, a static web page is far from the desired solution. This leaves the options of a Java web servlet or a Python web application. Both of these are good options, however, running a Python interpreter inside the Java Virtual Machine(JVM) is a much slower process than invoking Java classes from Python. This can be done using Jython, a Python implementation for Java that allows developers to both invoke Java from Python and vice versa [JBW+10]. An alternative solution is to run shell commands directly through the Python interpreter in order to run the Java classes. This is the chosen route of action because no information needs to be passed between the two platforms that cannot be sent as command-line arguments when running the Java code.

Now that this has been decided, a Python web framework must be chosen for the development of this GUI. The CherryPy web framework was selected ahead of the likes of Django and Pylons due to its simplicity and pythonic interface [Hel07]. For an appealing web design that is easy to create, a templating language must be used to embed Python code. This will be done using the Mako template library[2]. Mako is a leading templating library that is used by some major websites, such as `python.org` itself. Mako was chosen ahead of others such as Genshi, which is an XML-based templating engine, as its syntax is much more similar to that of normal Python code. Mako templates also support inheritance as can be seen in Listing C.2 provided in Appendix C. HTML5, CSS and the JavaScript library jQuery make up the rest of the web development environment.

As designed, the web application has to carry out three functions, corresponding to use cases 2, 3 and 4, as described in Section 3.2. Users must be able to **search** Twitter, **analyse** stored results and **view** the top tweeted operating systems and software. The first of these tasks, the search, is carried out using the Java implementation of integrating Twitter's Search API. Python's `subprocess` module allows shell commands to be executed inside the Python interpreter. This feature is exploited to run the `uk.ac.manchester.cs.patelt9.twitter.SearchAPI` class, its implementation details explored in Section 4.1.2, and check its output in the shell. This output is piped through to the web application and finally displayed on the web page. This is shown in Chapter 5 when discussing the results of the project.

The user may then opt to follow through the entire feature extraction process. By restricting the search to a fairly small number of tweets, this process is heavy in terms of time consumption. By opting to do so, the user allows the application to check the sentiment of each of the tweets, and then follow through the feature extraction process described in Section 4.2. At the end of this extraction, all the new information is returned in the form of a list, ready for display on the web page.

The next task is to show the user the results that have already been stored in the database. As per the design, this is done by showing the user a drop down menu consisting of all of the software that exists in the database. Upon selecting from the list, the application uses the aggregation methods described in the previous section to retrieve sentiment data. These are then used as input to the Google Chart Tools API, which creates pie charts displaying percentages for each distinct sentiment value.

Finally, the web application's home page shows a list of the top ten most tweet operating systems and software as links to the charts of these tools. This is again done using the aggregation methods set up in the previous stage of implementation.

4.5 Testing

Due to the agile approach taken in the development of the SWOT system, testing was carried out iteratively throughout the project. This was done in the form of unit and integration tests. Unit testing refers to tests that verify code modules or class function properly. These tests were performed throughout implementation to ensure each core stage was fully functional, particularly in the feature extraction phase. For example, in the price extraction process, the string "$8 million " should return the full string as a price whereas "8 million" should yield a negative price match.

[2] `http://www.makotemplates.org/`

Integration testing is performed when any two fully tested units are combined to create a larger structure. This was also carried out throughout development but was most apparent when implementing SWOT's web application. The web application required the integration of every other Python function developed as well as the Java implementation of the Twitter search use case. For optimal user experience, these independent functions must integrate seamlessly to appear transparent to the user.

System tests were performed towards the end of the implementation of the system in order to verify the initial requirements stated in the design phase (see Chapter 3) have been met. In SWOT's case, this is essentially ensuring all of the specified features have been extracted from tweets where possible, and further details have been provided on this stage in Chapter 5, where results have been detailed.

Chapter 5

Results

This chapter details the results of the project, in terms of the accuracy of the feature extraction, along with views of the user interface.

5.1 Feature Extraction

The analysis of results is a semi-automated process carried out by checking which tweets were said to contain software and which did not. In total, 3268 tweets were retrieved from Twitter and stored in the MySQL database. These have all been through the feature extraction process, and 1168 of these are said to mention software or operating systems, leaving 2100 that do not. Of these tweets, roughly 10%, i.e. 350, were manually tagged purely with software names, for the purpose of evaluating the system. Comparing these results produces the accuracy measures shown in Tables 5.1 and 5.2. While 350 tweets were tagged, the Table 5.1 states a total of 367. This is because many tweets contain more than one software name.

A precision of 79% is reasonable as this means that 79% tweets were correctly found to have contained software. 68% recall is fairly low as this means 32% of the tweets retrieved from Twitter are actually irrelevant in the sense that they are actually unrelated to software. This could well be an issue with the keywords used, but also perhaps more likely due to the ambiguity associated with many software names, such as *Blender*, which is a 3D computer graphics product as well as a kitchen appliance. The 84% specificity suggests SWOT is quite successful at identifying negative software matches when processing tweets.

True Positives	True Negatives	False Positives	False Negatives	**Total**
117	162	32	56	**367**

Table 5.1: Results from comparing manual and automated tagging

Precision	Recall	Specificity	**F-measure**
0.79	0.68	0.84	**0.73**

Table 5.2: Precision, recall, specificity and F-measure

5.2 GUI

The GUI for this system has been implemented to the design requirements stated in Section 3.3.3. All pages display a navigation bar allowing access to the Tweet Retrieval subsystem as well as the analysis pages. The home page displays the top ten tweeted software as found by the application. This can be seen in Figure 5.1. The elements in this list are clickable hyperlinks to their respective analysis pages.

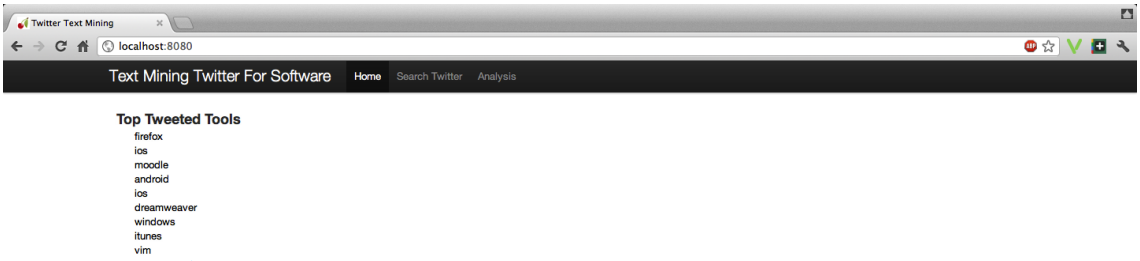

Figure 5.1: The web application's home page

The search page asks users to enter a single query term as shown in Figure 5.2. This should ideally be the name of some software, operating system or programming language. After completing this search, the page displayed in Figure 5.3 shows all retrieved tweets corresponding to the given query term, after being sent through the feature extraction process.

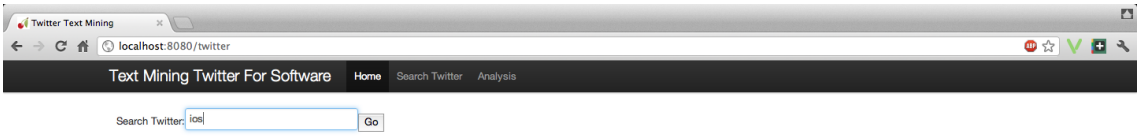

Figure 5.2: The web application's search page

The final requirement is to display the key information to the user. This involves displaying a pie chart representing the sentiment of tweets mentioning the selected software, and is shown in Figures 5.4 and 5.5.

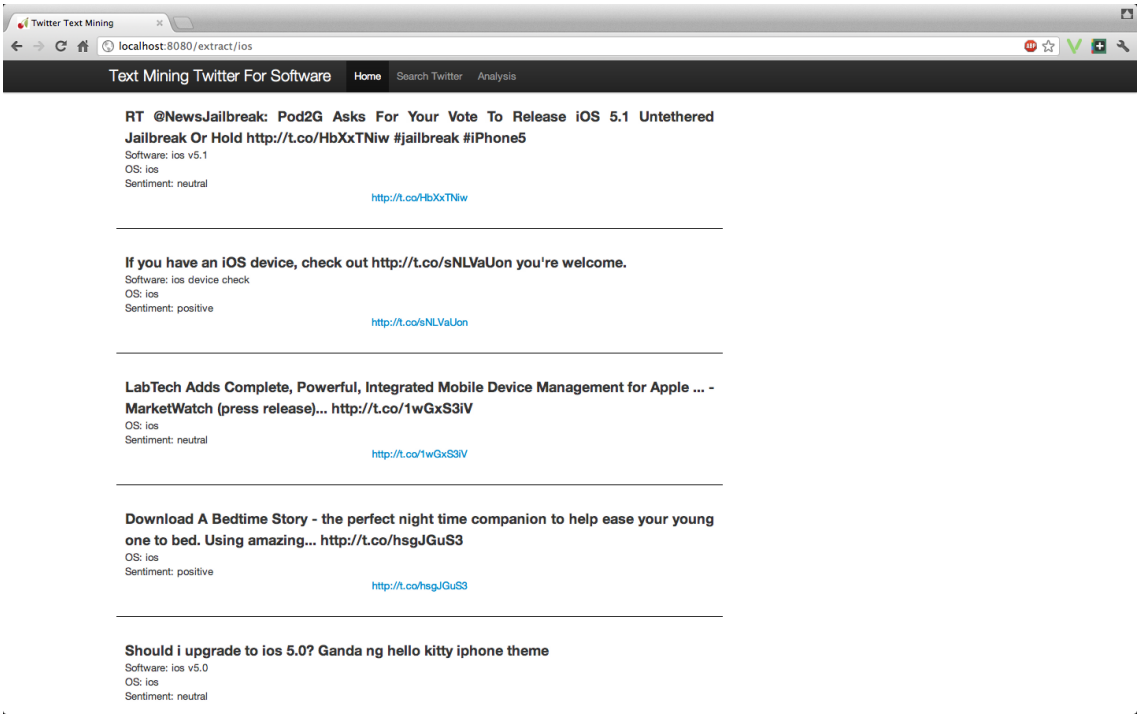

Figure 5.3: The web application's extracted features page

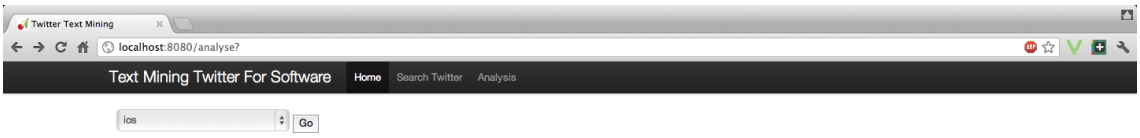

Figure 5.4: The web application's analysis dropdown menu

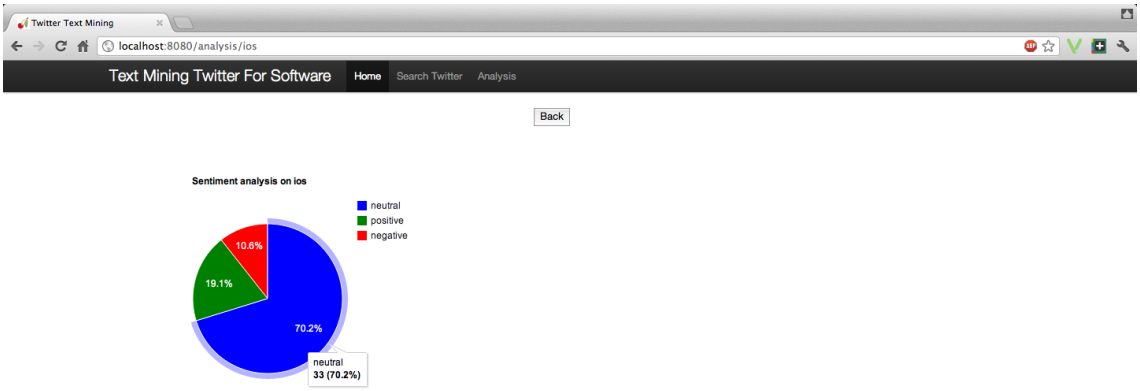

Figure 5.5: The web application's analysis page for a specified piece of software

Chapter 6

Evaluation

With implementation and analysis complete, one can now identify and evaluate the key successes and shortcomings of this project. These evaluations have been performed on the basis of the following questions and has been carried out by means of 5 independent user evaluations of the software.

- Does the SWOT system work?
- Is the information found novel and interesting?
- Is the system easy to use?

SWOT was shown to work to an accuracy of 73% when measuring against software names. This is a relatively low performance rating as over larger sets of data the number of inaccurate results will be much greater. This is mainly down to the recall of the system. There are many situations where references to software have been missed in tweets. There is also a large set of tweets where they are in fact not related to software or explicity mention their names. However, this may be caused by the ambiguous nature of the terms in the dictionary and keyword database. Improving these results may require more linguistic rules to better analyse text.

The information found by the system is mainly sentiment data. While this may be interesting to certain parties, user opinion appears somewhat indifferent. Sentiment data would be interesting to the developers, and users looking for new software. However, while SWOT recognises software that is not present in the dictionary, these software are not explicitly flagged as new when they first appear in tweets. Also, unless a user specifically searches for a new piece of software, it is unlikely to appear in many tweets present in the database. This should only be an issue in the early stages after deployment, as over time, increasing numbers of tweets will contain this software particularly after it has been added to the dictionary and is used to filter on Twitter's Streaming API. On the other hand, with more popular software, the sentiment information may not be particularly interesting to most users because the general perception towards it is likely to be common knowledge. However, SWOT can still be used to verify this information.

Finally, the system's usability is one of the key issues in this project. The web application has a very simple, straightforward interface. Some users found this to be more appealing as it looks tidy, however, others felt it was minimalistic, particularly the home page, though easy to use. Users also suggested the analysis page required more charts and extra information, such as the tweets themselves.

Chapter 7

Conclusions

This chapter discusses the author's reflections on the success and conclusions of the project. The report concludes with suggestions for further work.

7.1 Achievements

With respect to the initial aims and objectives of this project, the final outcome is fairly successful. The system has completed the objectives of retrieving tweets from Twitter, extracting the features detailed in Section 3.3.2 from them, and then displaying the aggregated results to the user.

SWOT scores an F-measure of 0.73, which is a fairly good score when considering the complexity of named entity recognition in the domain of software names.

7.2 Reflections

In terms of the overall progress made over the course of the year I feel this project has been a success. It has been a steep learning curve and on that basis some good results have been achieved. However, in absolute terms, I think results could have been slightly better and this may be because of some mistakes I have made in the way I went about working on the project over the year. This is down to a few key issues.

Time management can be seen as one of the overriding causes of some of the shortfalls of this project. There were times when progress had completely stalled due to minor issues with implementation. I also feel I spent too much time on the first phase of the project. Priorities should have been placed on the second phase, where features were extracted from tweets.

I also feel as though my **preparation** may not have been sufficient, as I had overlooked a few key concepts when developing parts of the system. For example, I did not create a training set of data that would ultimately allow me to use a machine learning approach in my system and I think this may have resulted in a slightly lesser performing program, in terms of its accuracy.

However, as previously stated, there have been major strides over the year. I feel I have a much greater grounding in the core object oriented design and programming principles. As well as this, I have developed skills in many new technologies and concepts. This was the first time I have had to develop a multithreaded desktop environment, and I had never previously worked with Python, JSON, MongoDB or the HTTPS protocol. As such I am happy with the general progress made in this project.

7.3 Future Work

With extra time for the project, a few more functions could be implemented. Firstly, an interesting feature to extract from tweets would be the underlying reason for posting said tweet. This could, for example, be for the purpose of marketing, or simply a user reviewing the software. The ratio of these purposes could return some interesting results as to who tweets about which software.

Another function could be to associate certain words with software. For example, *download* could be used in many tweets mentioning software. If associations with large support and confidence can be determined, these words could be added to the set of keywords and improve the overall relevancy of the tweets retrieved from Twitter's APIs.

As well as this, some extra linguistic rules could be defined for the system to perform better in identifying software.

Bibliography

[Alc11] AlchemyAPI. Sentiment analysis. http://www.alchemyapi.com/api/sentiment/, October 21, 2011.

[API12] Twitter API. Getting started. https://dev.twitter.com/start, April 2, 2012.

[Ban11] K. Banker. *MongoDB in Action*. Manning Pubs Co Series. Manning Publications, 2011.

[BLK09] Steven Bird, Edward Loper, and Ewan Klein. *Natural Language Processing with Python*. O'Reilly Media Inc, 2009.

[BMZ11] Johan Bollen, Huina Mao, and Xiaojun Zeng. Twitter mood predicts the stock market. *Journal of Computational Science*, 2(1):1 – 8, 2011.

[CCMS10] Courtney D Corley, Diane J Cook, Armin R Mikler, and Karan P Singh. Text and structural data mining of influenza mentions in web and social media. *International Journal of Environmental Research and Public Health*, 7(2):596–615, 2010.

[Cha97] Eugene Charniak. Statistical techniques for natural language parsing. *AI Magazine*, 18:33–44, 1997.

[CLi12] CLiPS. Pattern.en sentiment. http://www.clips.ua.ac.be/pages/pattern-en#sentiment, April 6, 2012.

[Cro12] Douglas Crockford. Introducing json, April 28, 2012.

[Cul10] Aron Culotta. Detecting influenza outbreaks by analyzing twitter messages. *CoRR*, abs/1007.4748, 2010.

[FD95] R. Feldman and I. Dagan. Kdt - knowledge discovery in texts. In *Proc. of the First Int. Conf. on Knowledge Discovery (KDD)*, pages 112–117, 1995.

[GBH09] Alec Go, Richa Bhayani, and Lei Huang. Twitter sentiment classification using distant supervision. *Processing*, pages 1–6, 2009.

[GL09] Vishal Gupta and Gurpreet S Lehal. A survey of text mining techniques and applications. *Journal of Emerging Technologies in Web Intelligence*, 1(1):60–76, 2009.

[Gri08] Seth Grimes. Unstructured data and the 80 percent rule. Carabridge Bridgepoints, 2008.

[GS96] Ralph Grishman and Beth Sundheim. Message understanding conference - 6: A brief history. In *Proceedings of the International Conference on Computational Linguistics*, 1996.

[Hea99] Marti A. Hearst. Untangling text data mining. In *Proceedings of the 37th annual meeting of the Association for Computational Linguistics on Computational Linguistics*, ACL '99, pages 3–10, Stroudsburg, PA, USA, 1999. Association for Computational Linguistics.

[Hel07] Sylvain Hellegouarch. *CherryPy Essentials: Rapid Python Web Application Development Design, develop, test, and deploy your Python web applications easily*. Packt Publishing, 2007.

[HNP05] Andreas Hotho, Andreas Nürnberger, and Gerhard Paaß. A brief survey of text mining. *LDV Forum - GLDV Journal for Computational Linguistics and Language Technology*, 20(1):19–62, May 2005.

[Hun07] John D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, May-Jun 2007.

[JBW+10] Josh Juneau, Jim Baker, Frank Wierzbicki, Leo Soto, and Victor Ng. *The Definitive Guide to Jython: Python for the Java Platform*. Apress, Berkely, CA, USA, 1st edition, 2010.

[KV05] M. Krallinger and A. Valencia. Text-mining and information-retrieval services for molecular biology. *Genome Biol*, 6(7), 2005.

[Lin12] Linguamatics. Text mining solutions for news & social media. `http://www.linguamatics.com/welcome/solutions/news_rss_social_media_analysis.html`, April 20, 2012.

[LMRS07] Witold Litwin, Riad Mokadem, Philippe Rigaux, and Thomas Schwarz. Fast ngram-based string search over data encoded using algebraic signatures. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 207–218. VLDB Endowment, 2007.

[MRS08] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[PD11] Michael J. Paul and Mark Dredze. You are what you tweet: Analyzing twitter for public health. In Lada A. Adamic, Ricardo A. Baeza-Yates, and Scott Counts, editors, *ICWSM*. The AAAI Press, 2011.

[PL08] Bo Pang and Lillian Lee. Opinion mining and sentiment analysis. *Foundations and Trends in Information Retrieval*, 2(1-2):1–135, 2008.

[Pol06] Tamara Polajnar. Survey of text mining of biomedical corpora. June 2006.

[PP10] Alexander Pak and Patrick Paroubek. Twitter as a corpus for sentiment analysis and opinion mining. In *Proceedings of the Seventh conference on International Language Resources and Evaluation (LREC'10)*, Valletta, Malta, May 2010. European Language Resources Association (ELRA).

[Rea05] Jonathon Read. Using emoticons to reduce dependency in machine learning techniques for sentiment classification. In *Proceedings of the ACL Student Research Workshop*, ACLstudent '05, pages 43–48, Stroudsburg, PA, USA, 2005. Association for Computational Linguistics.

[Sar08] Sunita Sarawagi. Information extraction. *Found. Trends databases*, 1(3):261–377, March 2008.

[Ski08] Diane J. Skiba. Nursing education 2.0: Twitter & tweets. *Nursing Education Perspectives*, 29(2):p110 – 112, 2008.

[SLW11] Inderjeet Singh, Joel Leitch, and Jesse Wilson. Gson user guide. `https://sites.google.com/site/gson/gson-user-guide`, October 15, 2011.

[Sod99] Stephen Soderland. Learning information extraction rules for semi-structured and free text. *Machine Learning*, 34:233–272, 1999. 10.1023/A:1007562322031.

[Twi12] Twitter. Twitter turns six. `http://blog.twitter.com/2012/03/twitter-turns-six.html`, March 21, 2012.

[WH98] Mark Warschauer and Deborah Healey. Computers and language learning: an overview. *Language Teaching*, 31(02):57–71, 1998.

Appendix A

Dictionary of Software and Keywords

Apache	Mozilla	Activision
Infinity Ward	Blizzard	EA
Rockstar Games	HP	Codemasters
Valve	Adobe	Blackberry
Cisco	SlingMedia	Maxis
Rarlabs	VideoLan	ATI
Atari	Symantec	Nvidia
Oracle	Apple	Microsoft
Google		

Table A.1: Dictionary of companies

WebOS	Android	Bada
Symbian	Linux	Fedora
Ubuntu	Windows	iOS
OS X	MacOS	FreeBSD

Table A.2: Dictionary of operating system names

ZeuAPP	Bitcoin	vtiger CRM
ReOS	SugarCRM	OrangeHRM
Ebase	Dolibarr ERP/CRM	Bonita Open Solution
Adempiere	bookyt	Compiere
FrontAccounting	GnuCash	Grisbi
HomeBank	jFin	JFire
JGnash	JQuantLib	KMyMoney
LedgerSMB	MibianLib	Mifos
Octopus Micro Finance Suite	Openbravo	OpenERP
Postbooks	QuickFIX	QuickFIX/J
SQL Ledger	Tryton	TurboCASH
WebERP	refbase	Koha
NewGenLib	OpenBiblio	PMB
SimPy	CellProfiler	ImageJ
Endrov	Jmol	Molekel
MeshLab	PyMOL	QuteMol
RasMol	Avogadro	Ascalaph Designer
GROMACS	LAMMPS	MDynaMix
NAMD	Chemistry Development Kit	JOELib
OpenBabel	P-GRADE Portal	OpenCog
OpenCV	AForge.NET	ROS
TREX	CMU Sphinx	Emacspeak
Festival Speech Synthesis System	NVDA	Text2Speech
NonVisual Desktop Access	ESpeak	Dasher
Gnopernicus	Virtual Magnifying Glass	OpenAFS
Eucalyptus	AppScale	FusionCharts
ParaView	Orange	RapidMiner
Scriptella ETL	Weka	jHepWork
Konstanz Information Miner	KNIME	ELKI
Lucene	Solr	Xapian
Nutch	CloverETL	Talend
Pentaho	SpagoBI	Limesurvey
OpenX	RSS Bandit	RSSOwl
Akregator	Liferea	Ekiga

Table A.3: Dictionary of software

FreePBX	FreeSWITCH	Jitsi
QuteCom	sipX	Slrn
FreeNX	OpenVPN	rdesktop
VNC	RealVNC	TightVNC
UltraVNC	xrdp	HTTrack
Wget	Apache Cocoon	AWStats
BookmarkSync	CougarXML	curl-loader
HTTP File Server	Distributed ICDL Crawler	Crawley Framework
lighttpd	nginx	NetKernel
Piwik	Qcodo	Web-Developer Server Suite
XAMPP	Zope	Oxwall
Liferay	Sun Java System Portal Server	uPortal
Apache Axis2	Apache Geronimo	GlassFish
CORBA	JacORB	Jakarta Tomcat
JBoss	ObjectWeb JOnAS	OpenSplice DDS
SmartVariables	OpenLDAP	JXplorer
openVXI	YaCy	Gnaural
DoceboLMS	eFront	GCompris
IUP	Moodle	Omeka
Sakai	Chamilo	openSIS
ATutor	ILIAS	Kiten
KVerbos	KTouch	KGeography
KEduca	openlp.org	BibleDesktop
BibleTime	Xiphos	SWORD Project
SwordBible	Go Bible	jSword
MacSword	Marcion	Eye of GNOME
F-spot	Gqview	Gthumb
imgSeek	Kphotoalbum	Dream DRM Receiver
KToon	Synfig	K-3D
OpenFX	Seamless3d	Pencil Animation
SWFTools	Avidemux	AviSynth
Blender	Cinelerra	CineFX
Jahshaka	DScaler	DVD Flick

Table A.4: Dictionary of software

DVDx	Kaltura	Kino
Kdenlive	OpenShot Video Editor	PiTiVi
VirtualDub	VirtualDubMod	Celtx
KeePass	Password Safe	TeamLab
Project.net	KAddressBook	Kontact
KOrganizer	Novell Evolution	OpenSync
Rachota Timetracker	Bugzilla	Mindquarry
Redmine	Trac	Open Scene Graph
OpenSCDP	CodeSynthesis XSD	CodeSynthesis XSD/e
xmlbeansxx	Flex lexical analyser	Kodos
phpCodeGenie	ĥxt2regex$	SableCC
Autoconf	Automake	Xnee
Memtest86	JSystem	GNU Debugger
ClamAV	ClamWin	Gateway Anti-Virus
AVG antivirus	Winpooch	MyDLP
GnuPG	KGPG	Seahorse
GnuTLS	OpenSSL	CrossCrypt
FreeOTFE and FreeOTFE Explorer	Iptables	Coyote Linux
Firestarter	IPFilter	ipfw
IPCop	IPFire	M0n0wall
PeerGuardian	pfSense	SmoothWall
Shorewall	Untangle	Vyatta
Zentyal	Lsh	OpenSSH
PuTTY	Cyberduck	Agorum
Anti-Spam SMTP Proxy	Balsa	Bogofilter
Citadel/UX	Claws Mail	Dada Mail
DSPAM	Enigmail	Exim
Fdm	Fetchmail	FreePOPs
FUDforum	Getmail	GNUMail
Gnus	Gnuzilla	GPGMail
GroupServer	Hypermail	I-sense
IlohaMail	Internet Messaging Program	Libremail
GNU Mailman	MailScanner	Mailx

Table A.5: Dictionary of software

Majordomo	MH Message Handling System	MIMEDefang
Movemail	Mozilla Mail	Mozilla Thunderbird
Mutt	NeoMail	Open-Xchange
POPFile	Qpopper	Roundcube
Scalix	SimpleMail	Smail
Smartlist	SpamAssassin	Spicebird
SquirrelMail	Sylpheed	Sympa
Uebimiau	Universal village collaboration suite	UW IMAP
Vpopmail	Xuheki	Yet Another Mailer
YPOPs!	Zarafa	Zimbra
Adium	AMSN	Ayttm
BitlBee	Bombus	Bombusmod
Centericq	Climm	Coccinella
Emesene	Fama IM	Gabber
Gajim	Instantbird	JabberMixClient
Jimm	JWChat	Kadu
Kopete	Meetro	Miranda IM
Monal	Naim	Neustradamus
Pandion	Pidgin	Proteus
Retroshare	SFLphone	Tapioca
TerraIM	Tkabber	Tmsnc
TorChat	Zephyr	uTorrent
Skype	Sopcast	Spotify
Google Chrome	Steam	TextMate
Text Editor	TextEdit	nedit
gedit	BitLord	bittorrent
iTunes	AVG	MS Office
f.lux	WinRar	VLC
Radeon	Daemon tools	Dreamweaver
emacs	Firefox	

Table A.6: Dictionary of software

Appendix B

Bing API Integration

```
'''
bing.py

Created on Dec 28, 2011

@author: Tariq Patel
'''
try:
    import json
except ImportError:
    import simplejson as json
from urllib import urlencode

from httplib2 import Http

class BingSearch:
    def __init__(self, app_id):
        self.app_id = app_id

    def search(self, query, source_type=None, api_version=None, max_results
        =None, **kwargs):
        kwargs.update({
            'AppId': self.app_id,
            'Version': api_version or '2.2',
            'Query': query,
            'Sources': source_type or 'Web',
            'Web.Count': max_results or '15'
        })

        _response_, contents = Http().request('http://api.bing.net/json.
            aspx?' + urlencode(kwargs))
        try:
            return _BingResponse(json.loads(contents)['SearchResponse']['
                Web'])
        except ValueError:
            raise ServerError('Could not connect to Bing')

class _BingResponse(dict):
    def __init__(self, resp):
        dict.__init__(self, resp)
```

```
    def get_results(self):
        return self['Results'] # Returns only max_results number of results

    def __len__(self):
        return self['Total']

class ServerError(Exception):
    def __init__(self, *args, **kwargs):
        Exception.__init__(self, *args, **kwargs)
```

bing.py

Appendix C

Web Application Python Code

```
'''
__main__.py
Web Application implementation

@author: Tariq Patel
'''
import subprocess
from sys import exit

import cherrypy
from mako.template import Template
from mako.lookup import TemplateLookup
import routes

import argument_parser as argparse
from database_connector import MongoConnector, SQLConnector
import java_utils as java
from search import ImgCreator
import ssh
from tagger import TweetTagger
from twittersentiment import bulk_analysis
from web import JavaScript

class WebApp(object):
    ''' INITIALISATION '''
    def __init__(self, dirs, auth):
        self._tmpl = TemplateLookup(directories=dirs)
        self._nav = {
                     'title':'Text Mining Twitter For Software',
                     'nav':'nav.html',
                     'results':'../results',
                     'search':'../twitter',
                    }
        self._auth(user=auth.user, passwd=auth.password, db=auth.db,
                   sport=auth.port, mport=auth.mongoport)

    def _auth(self, user, passwd, db, sport=3306, mport=27017):
        try:
            self._sql = SQLConnector(host='127.0.0.1',
                                     user=user,
                                     passwd=passwd,
```

```
                                    db=db,
                                    port=int(sport))
            self._mongo = MongoConnector(host='localhost',
                                         db=db,
                                         port=int(mport))
        except Exception, e:
            exit(e)
        self._tagger = TweetTagger(sql=self._sql, mongo=self._mongo)
        self._imgc = ImgCreator(mongo=self._mongo)
    ''' END INITIALISATION '''

    ''' TEMPLATE HELPER FUNCTIONS '''
    def _get_template(self, file, **kwargs):
        kwargs.update(self._nav)
        return self._tmpl.get_template(file).render(**kwargs)

    def _template(self, body):
        return Template('<%inherit file="base.html"/>'+body,lookup=self.
            _tmpl).render(**self._nav)
    ''' END TEMPLATE HELPER FUNCTIONS '''

    ''' TOP TWEETED TOOLS '''
    @cherrypy.expose
    def index(self):
        return self._get_template('index.html', tools=self._mongo.top_ten()
            )
    ''' END TOP TWEETED TOOLS '''

    ''' TWITTER SEARCH API '''
    @cherrypy.expose
    def twitter(self):
        return self._get_template('search.html', action='../twittersearch')

    @cherrypy.expose
    def twittersearch(self, query):
        return self._template(body='Searching Twitter...' + JavaScript.
            redirect('twitter/%s' % query))

    @cherrypy.expose
    def tweets(self, query):
        tweets = java.search_twitter(query)
        if not len(tweets):
            return self._template(body='No tweets were found')
        return self._template(body='Extracting features...' + JavaScript.
            redirect('../extract/%s' % query))

    @cherrypy.expose
    def extract(self, query): # Routed as /extract/:query
        bulk_analysis(sql=self._sql, keyword=query)
        tweets=self._tagger.tag(keyword=query)
        if len(tweets):
            return self._get_template('tweet.html', tweets=tweets)
        return self._template('No tweets to analyse')
    ''' END TWITTER SEARCH API '''

    ''' ANALYSIS '''
    @cherrypy.expose
```

```
    def results(self): # This page loads data for analysis
        return self._template(body='Retrieving data'+ JavaScript.redirect('
            ../analyse'))

    @cherrypy.expose
    def analyse(self):
        elements = self._mongo.find_all()
        if not len(elements):
            return self._template(body='No software has been found yet')
        return self._get_template(file='show_results.html', action='../
            analysis', elements=elements)

    @cherrypy.expose
    def analysis(self, software):
        raise cherrypy.HTTPRedirect('analysis/%s' % software)

    @cherrypy.expose
    def aggregate(self, name): # routed as /analysis/:name
        data, col = self._imgc.web_query(name)
        if not len(data):
            return self._template(body='That software has not been found')
        return self._get_template(file='google-charts.html', button='../
            analyse', heading=name, data=data, colours=col)
    ''' END ANALYSIS '''


def setup_routes(args):
    w = WebApp(dirs=['web'], auth=args)
    d = cherrypy.dispatch.RoutesDispatcher()
    d.connect('index', '/', controller=w, action='index')
    d.connect('main', '/:action', controller=w)
    d.connect('main-1', '/:action/', controller=w)
    d.connect('res', '/analysis/:name', controller=w, action='aggregate')
    d.connect('res-1', '/analysis/:name/', controller=w, action='aggregate'
        )
    d.connect('search', '/twitter/:query', controller=w, action='tweets')
    d.connect('search-1', '/twitter/:query/', controller=w, action='tweets'
        )
    d.connect('extract', '/extract/:query', controller=w, action='extract')
    d.connect('extract-1', '/extract/:query/', controller=w, action='
        extract')
    return d

if __name__ == '__main__':
    import os.path
    config = {
              '/':{
                   'request.dispatch': setup_routes(args=argparse.main()),
                   'tools.staticdir.root': os.path.dirname(os.path.abspath(
                       __file__)) + "/web"
                  },
              '/css':{
                      'tools.staticdir.on': True,
                      'tools.staticdir.dir': 'css'
                     },
              '/js':{
                     'tools.staticdir.on': True,
```

```
                    'tools.staticdir.dir': 'js'
                }
            }
    cherrypy.tree.mount(None, config=config)
    cherrypy.engine.start()
    cherrypy.engine.block()
```

Listing C.1: Web application implementation

```
<!DOCTYPE html>
<html>
    <head>
        <title>Twitter Text Mining</title>

        <link rel="stylesheet" type="text/css" href="/css/bootstrap.min.css
            " />
        <%block name="style"/>
        <link rel="stylesheet" type="text/css" href="/css/bootstrap-
            responsive.min.css" />

        <script type="text/javascript" src="http://ajax.googleapis.com/ajax
            /libs/jquery/1.4.2/jquery.min.js"></script>
        <script type="text/javascript" src="/js/libs/modernizr-2.5.3-
            respond-1.1.0.min.js"></script>
        <%block name="script"/>
    </head>

    <body>
    <%include file="${nav}" />
    ${self.body()}
    </body>
</html>

<%def name="js(path)">
    <script type="text/javascript" src="${path}"></script>
</%def>
```

Listing C.2: The base.html base class for all web pages