Map the Debate

Understanding the web's response

Author: Joseph Root <jsr08@ic.ac.uk>
Supervisor: Francesca Toni <f.toni@ic.ac.uk>

June 2011

Imperial College London

ACKNOWLEDGEMENTS

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

ABSTRACT

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

CONTENTS

PART I | ANALYSIS

1

INTRODUCTION

1.1 Motivation

From women's rights to civil rights, the influence of public opinion on government policy has been pivotal. Within a healthy democracy the voice of the electorate should be heard and recognised by those chosen to represent them. Throughout history platforms have often been provided for public opinion to be made known, from the early public forums of Greece and Rome, to speaker's corner and the house of commons today. Providing a means for people to express their opinion enables them to both challenge and shape the direction their elected governments take. Finding ways of gathering and understanding this opinion has increasingly proven fundamental if a government wishes to be successful.

Current methods of measuring opinion are largely statistical, with methods such as polling looking at the opinion of a sample group, before using their results to make further predictions. These can be very accurate, however their small sampling rates mean that figures can often be askew. Furthermore polling is both costly and time consuming to conduct, and thus can neither be used to find opinion on breaking news or on a variety of topics. None the less, as methods for measuring public opinion have increased both in accuracy and detail, politicians and policy makers are starting to look to them not only for affirmation of their policies, but for guidance and new initiative.

As the web has become more prevalent throughout society, it is increasingly becoming a platform for discussion and opinion. The initial growth of blogging demonstrated the web's ability to serve as a forum for debate and opinion. However the technical knowledge required to start a blog, alongside the time required to write a post meant that adoption was limited. In the past two years we have seen the rise of micro-blogging (essentially 140 character blog posts) through services such as Twitter. These have seen unprecedented levels of adoption, with Twitter's 200 million users posting 25 billion micro-blog posts in 2010. Due to the simple nature of writing short posts, micro-blog discussion tends to break quickly around news topics, and offers genuine insight into public opinion surrounding news topics.

This project hopes to utilise the growth of publicly available opinion on the web, using it as a source upon which new methods for analysing and measuring public opinion can be built. In particular the project will focus on understanding sentiment on micro-blogging services such as Twitter.

1.2 Contributions

1. Ruby implementation of a sentiment analysis engine based upon current research. The engine should be able to correctly identify sentences containing sentiment, and classify the sentiment as positive or negative. Different implementations should be tested and compared to determine which algorithms and techniques work best.
2. An algorithm for classifying sentiment as a range of emotion and feeling, rather than just a score along a scale of positive to negative. The algorithm should be implemented in Ruby and included in the sentiment analysis engine.
3. Research optimisations for current algorithms, in order to tailor the sentiment analysis engine for micro-blog posts from services such as Twitter. These optimisation should be implemented within the engine.
4. Research optimisations for current algorithms, in order to better facilitate the understanding of Politically focussed micro-blog data. These optimisation should be implemented within the engine.
5. A Ruby based Twitter module to store and classify live data from Twitter.
6. Visualisations should be designed and implemented to help better understand the data and classification results.

2

BACKGROUND

From its early forums through to the 'social web' of today, the Internet has served as a continually expanding platform for discussion. The result has been an explosion in the amount of readily available, computer-formatted textual opinion. With this growth has come an increasing desire to computationally understand the wealth of opinion now so easily accessible. Combining elements of linguistics, natural language processing and machine learning, this field of exploration has come to be known as *opinion mining* or *sentiment analysis*. In the following chapter we will first briefly examine Twitter as a backdrop to our discussion on sentiment analysis. We will then go on to explore the general problems posed by sentiment analysis along with the common approaches and solutions taken in addressing them. In sections 2.2.2 - 2.2.4 we will discuss in detail the areas and methods of sentiment analysis which will bear relevance to this project's Twitter-based setting. In section 2.4 we will explore emotion in general, particularly looking at its scope and ways of classifying it. Finally in section 2.5 we shall discuss our project's choice of programming languages and tools.

2.1 Twitter

Twitter is a social-networking web-service. It enables users to post and read 140 character messages known as *tweets*. A user's *timeline* serves as a publicly viewable history of their tweets. Furthermore if someone chooses to *follow* another user, they will be notified of changes to that user's timeline. This simplicity has seen Twitter's user-base rapidly expand, with over 200 million active users today. From football transfers to revolutions Twitter has become the go-to service for spreading news quickly and efficiently.

Since its launch in 2006, certain protocols have emerged from within the Twitter community. These have been embraced by Twitter, enabling it to serve not only as an efficient platform for spreading news, but also as a rich and sophisticated medium for conversation. Notable protocols include:

Hashtags enable users to tag their tweets with any word or combination of characters they deem appropriate. Although this may seem basic at first, through common hashtags, it enables users to take part in a community-wide discussion. For example, during the recent voting reform referendum, the hashtags '*#yes2av*' and '*#no2av*' were used to form a debate on the strengths and weaknesses of the Alternative Vote.

Mentions allow users to reference other users in their tweets. Furthermore if a user is mentioned in a tweet, Twitter will notify the mentioned user. Through this, Twitter users can take part in a direct conversations with one or more other users. For example, if we wanted to ask Stephen Fry a question, we could tweet '*what are you eating for breakfast @stephenfry?*'.

Re-tweets give users the ability to re-post other users' tweets in their own timeline. This simple feature has had a significant impact on Twitter's ability to facilitate the rapid spread of news. For example in 2009 when the US Airways flight 1549 crash landed in the Hudson river, rapid re-tweeting of an amateur photo meant the news broke on Twitter far earlier than it did within the media at large. This has continued to be true for many more notable events such as the recent North-African revolutions.

Links have always been the popular subject of tweets, however the introduction of link-shorteners has changed the way in which they are posted. In freeing up characters by shortening a URL, users now have the option to describe or comment on the link they are tweeting. This has enabled users to engage in deeper conversation on content they have viewed online, and has neatly allowed Twitter's viral nature to better merge with its community's desire for debate.

Through Twitter's RESTful API [1], this rich resource of live news and debate will serve as the project's main data source.

2.2 Sentiment analysis

Sentiment analysis as a field, is the exploration of how we can computationally understand opinions expressed within a body of text. In order to do this, we must first define a computational structure for expressing opinions. In general (1) this is done by breaking an opinion down into four parts. Firstly we must determine the opinion's focus of discussion, also known as its *topic* [2]. This in practise can encompass anything from Government policy to mobile phone battery life. Often opinion is not necessarily that of the author, but of a referenced person or group, therefore it is important to determine the opinion's *holder*. Along with this it is also often necessary to determine the *time* at which the opinion was expressed. Finally,

[1] RESTful APIs allow developers to retrieve, modify, create and delete data by making get, post and delete HTTP requests to specified web addresses.

[2] This is more commonly referred to within literature as an opinion's *feature*, however to avoid later confusion with the machine learning term, we will use the term *topic*.

we hope to *classify* (or in some cases quantify) the opinion which has been passed. Leading research (2; 3) has typically focussed on discrete classification, such as deciding whether an opinion is positive, negative or neutral. A fifth *object* component is sometimes introduced for larger documents, which serves as an identifier for related topics. For example, within a phone review the majority of opinions may share the same object, in this case the phone, but focus on different topics such as battery life or call quality.

How do we computationally discover opinions and identify their parts? In general the approach can be loosely split into two components, *sentence-level classification* and *document-level classification*. Sentence-level classification determines whether a sentence expresses an opinion along with classifying that opinion if it exists. Furthermore if an opinion is found, sentence-level classification will try to determine its topic, holder and the time at which the opinion was cast. Document-level classification goes on to collate the sentence-level results, in order to form a general description of the document's sentiment. Both these approaches draw heavily upon machine learning techniques. It is important to note here however, a core criticism of the field. Linguists such as Chomsky (4) observe that rather than truly trying to understand and define the semantics of sentiment, the field takes a heavily statistical approach. This means that rather than determining sentiment by forming a semantic conclusion, the field uses a limited linguistic foundation to predict sentiment based upon experience. Nonetheless, redefining natural language processing and sentiment analysis is not within the scope of this project, and we shall proceed with the field's successfully tried and tested approaches.

As we shall discuss in more detail in chapter 6, only sentence-level classification is relevant to this project. Furthermore, methods for determining an opinion's holder and time are unnecessary and will not be discussed here. The remainder of this section will instead focus on the three relevant topics from within sentence-level classification. Firstly we shall explore what exactly an opinionated sentence is and how we can computationally determine this. Next we will look at common approaches to classifying sentiment, before finally examining how we determine the topic of an opinion. Before this however, we shall briefly outline the concepts and methods of *supervised learning* as this shall form the core for each of our classification problems.

2.2.1 Supervised learning

Supervised learning is a task within machine learning which infers a function from a set of training data. This approach is well suited to classification problems, and in our case is particularly relevant to classifying opinion and determining polarity. Thus, the remainder of this section will discuss supervised learning with respect to the classification of sentences.

2.2.1.1 *Defining the problem*

In both problems we want to find an approximate hypothesis function h for our actual function c. Both functions will map an input sentence $s \in S$, to a discrete classification $o \in O$, where S is the set of all possible sentences and O is the set of all possible classifications, for example $O = \{positive, neutral, negative\}$, such that:

$$h \approx c : S \to O \tag{2.1}$$

In order to find our best fit hypothesis function h, we will first need to determine a set of *features* for our sentences. Within machine learning, features are the attributes which best describe and discriminate our input data when trying to classify it. For example if we are trying to learn a function to decide whether we should play tennis or not, features might include humidity and sunlight. In essence we want to identify a list of the most useful features $f_1, f_2, \ldots, f_n$ for our sentences, such that:

$$h \approx c' : \langle f_1, f_2, \ldots, f_n \rangle \to O \tag{2.2}$$

Once a set of features has been chosen we can approximate h by training it. In order to find the perfect hypothesis function for classifying subjective functions, $h = c$, we would require knowledge of every single possible sentence along with its correct classification. Clearly we could never produce the set of all possible sentences, let alone determine every sentence's classification. Instead, we select a sample of training sentences $T \subseteq S$, and manually *label* each sentence $t \in T$ with a classification $l \in O$. This is our *training data* D, such that:

$$D = \{(t, l) : \forall t \in T \textit{ there exists a manually labelled classification } l\} \tag{2.3}$$

Given this training data we can now determine as accurate a hypothesis function as possible for classifying *all* sentences. There are numerous, largely statistical methods for training our hypothesis function. Each brings their own positives and negatives, and there has been extensive research (5) into which methods perform best for opinion based classification. We will discuss the most appropriate methods, features and training data for each classification problem in their respective parts.

2.2.1.2 *Common approaches*

2.2.2 Discovering opinion

In general opinion manifests itself either *explicitly* through *subjective* sentences and phrases, or *implicitly* through *objective* sentences and phrases. An objective sentence expresses factual information, whilst a subjective sentence expresses a mental or emotional state, such as a sentiment or belief. A subjective sentence such as, "*I love the NHS, it's bloody marvellous*", explicitly states an opinion. Similarly however, a sentence such as "*Lost my job due to recent Coalition cuts*" although objective, could also be considered an implicit opinion. This clearly poses a difficult challenge for classification, and as Mihalcea et al. (6) note, it is one which "has often proved to be more difficult than subsequent polarity classification". As observed by Liu (1) however, opinionated sentences tend to be a subset of subjective sentences. Due to this, the approaches for classifying them are similar and the terms are taken as interchangeable. This is referred to as *subjectivity classification*.

Subjectivity classification is typically achieved through a mix of supervised and unsupervised learning. In general, unsupervised learning is used to bootstrap a relatively small but accurate training set. The bootstrapped training set is then utilised to train a classifier. Numerous feature choices have been proposed for training subjectivity classifiers. We shall first examine some of the the more commonly used features, as discussed by Wiebe et al. (7):

Adjectives tend to be strong indicators of subjectivity, often serving as descriptions or qualifications of opinion. For example the adjectives in, "*the coalition cuts are **harsh** but **necessary***", are clear indications of subjectivity. As Wiebe et al. (7) observe a simple binary feature alone, noting the appearance of one or more adjectives, results in a classification accuracy of 56%.

Adverbs modify verbs, adjectives and phrases, for example "*they **usually** get things right*". Their presence is often an indicator of subjectivity, and although not as useful as adjective presence, their inclusion as a binary feature further improves classification rates. Wiebe et al. (7) suggest a binary feature noting the presence of any adverb other than *not*.

Pronouns are substitutions for nouns, for example *it* in place of an object. They are often minor indicators of subjectivity, and have been shown to marginally improve classification accuracy when included as a binary feature.

Adjective orientation and gradabilty tend to be further indicators of subjectivity. Essentially orientation notes whether an adjective encodes a desirable (e.g *beautiful*) or undesirable (e.g. *ugly*) state. The gradability of an adjective denotes the relative extent to which an adjective varies in strength from the norm. For example *"small"* and *"large"* have high gradability. As shown by Wiebe et al. (8), the presence of polarised, gradable adjectives is a strong measure of subjectivity and a useful feature.

Wiebe et al. (7) observed that using the first 3 techniques, coupled with cardinal numbers and a single document-level feature, resulted in classification rates of 71.2%.

But how can we identify these features within a sentence? Adjectives, adverbs and pronouns are all known as *parts of speech (POS)*. A word's POS can take on one of eight roles within a sentence: *verb*, *noun*, *pronoun*, *adjective*, *adverb*, *preposition*, *conjunction* and *interjection*. A word's part of speech is often determined by its position within the sentence. For example "*love*" can be a noun or a verb, dependant upon the context in which it is used. Below is an example of a sentence whose words have been *tagged* with their POS:

$$\underbrace{\text{She}}_{\text{pron.}}\ \overbrace{\text{likes}}^{\text{verb}}\ \underbrace{\text{big}}_{\text{adj.}}\ \overbrace{\text{snakes}}^{\text{noun}}\ \underbrace{\text{but}}_{\text{conj.}}\ \overbrace{\text{I}}^{\text{pron.}}\ \underbrace{\text{hate}}_{\text{verb}}\ \overbrace{\text{them.}}^{\text{pron.}} \tag{2.4}$$

2.2.2.1 *Part of speech tagging*

Given a phrase or sentence, *part of speech tagging* computationally determines each word's POS. This can be done in variety of ways. Typically basic implementations use a lexicon of words with their appropriate tags, or a more advanced dictionary such as WordNet[3]. In general these implementations are naive and often simply return a list of possibilities. More intuitive techniques tend to use machine learning to recognise patterns, or are built with a set of linguistic rules. We will discuss the merits of these techniques and their implementation in more detail in chapter 6. With a fully tagged sentence it is now possible to build a feature set based upon the relevant parts of speech.

2.2.2.2 *Use of supervised techniques*

As noted in our discussion of features, some adjectives are more useful in classifying subjectivity than others. Determining these adjectives, and in this case their polarity, would prove tedious if carried out by hand. Instead, Wiebe (9) suggests a supervised approach using a set of seed words and a large corpora of text. The corpora is examined for conjunctions, such as "*handsome* and *smart*", and disambiguations such as "*smart but cruel*". When a seed word is found within either scenario, its fellow word's polarity can be inferred. For conjunctions, if one of the words is known as positive, then the unknown word is likely to be positive also. The converse holds for disambiguations, where the unknown word is inferred to be the opposite of the known word. This technique enables the rapid building of a polarised adjective lexicon. It is particularly useful in domains which assign their own meaning to adjectives, for example *sick* is often a positive adjective within youth culture.

Building a training set significant enough for accurate subjectivity classification can often be time consuming. Liu (1) and Akkaya et al. (10) describe a supervised method for bootstrapping an initial training set. A high precision, low recall rule

[3] Wordnet is a detailed dictionary with additional levels of detail describing the semantic interlinking between words. It will be used throughout this project and shall be discussed in more detail in chapter 6.

based classifier, as originally proposed by Wiebe and Riloff (11), is used to build a small training set from a large corpora. The classifier does this by identifying strong and weak subjective clues within a sentence. If there are two or more strong subjective clues the sentence is classified as subjective. In order to determine objectivity, the sentences on either side are taken into account. If between them neither contain more than one strong and two weak clues, along with no strong clues in the analysed sentence, the sentence is considered objective. If the conditions for subjectivity and objectivity are not met, the classifier leaves the sentence unclassified. The use of supervised methods such as this and the lexicon builder described above are typical within the field. They provide simple and efficient ways of optimising the overall training process.

2.2.2.3 Present research and issues

Recent literature has also explored numerous improvements to the classic algorithm as described above. One such improvement of notable effect is *subjectivity word sense disambiguation (SWSD)*, originally presented by Wiebe et al. (12), and further refined by Akkaya et al. (10). SWSD tries to reduce the misclassification of objective words, and thus possibly the sentence, as subjective. These false hits often occur as a result of assuming that if a word exists within a subjective lexicon, it is being used in subjective sense. For example, *pain* is often used subjectively, however within, *early symptoms include body pain*, *pain* is used in an objective sense. SWSD attempts to eliminate this source of error. A subjective lexicon of words is built, and for each of its words, a classifier is trained. Given a potentially subjective word within a sentence, the classifier will label the word's sense as objective or subjective. The classifier is trained using a corpora of sentences whose subjective words have been labelled as either subjective or objective. The classifier is then used to ensure that all subjective words are used in their subjective sense. Using SWSD within subjectivity classification, Wiebe et al. (10) noted a 24% reduction in error against a classifier using the regular subjectivity lexicon when looking for subjective words.

Subjectivity classification is a well researched field, however current methods do pose problems. As is typically the case within supervised learning, the classifier's ability is significantly influenced by how representative its training set is of the input domain. Subjectivity classification, along with many other natural language approaches, is often extremely sensitive to the type of content with which it has been trained. This means that if one wants to build a subjectivity classifier for political speeches, the training corpora should be built from similar content, not for example from movie reviews. No fixed approach has been developed for this, and it is an issue we shall have to contend with during our implementation in chapter 6.

2.2.3 Classifying opinion

An opinionated sentence can express a diverse range of sentiment, and classifying this can prove difficult. Sentiment can be classified in numerous ways, for example "*I liked the tone of his speech, however I am uncertain of the proposals within it*", could be interpreted in any number of ways. At a phrase level, we might consider the first part to express some form of delight, while the latter expresses distrust. Of course, to a certain extent these are subjective, and more detailed emotional labels shall be discussed in section 2.4. A more broad classification might classify the first part as positive and the second part as negative. Developing methods for labelling a sentence's polarity has served as a focus for much of the research into classifying opinion. This field is referred to as *sentiment classification*.

But how do we determine sentiment? At first this may seem simple. For example "*I **love** the EU*" would typically be classified as positive, whilst "*I **hate** the EU's decentralisation of power*" would be negative. Clearly *love* and *hate* are strong indicators of polarity. Basic methods for classifying sentiment simply check whether any of the words within the sentence exist within a pre-defined polarity lexicon, and classify accordingly. If we explore increasingly complex phrases however, the problem becomes far less simple than simply identifying polarising words. Understanding the scope of negation can present challenges. For example the negative in "*not nice*", simply negates its neighbour, whilst in "*no one thinks that its good*", the ensuing negation spans the phrase. In certain scenarios negation words can even strengthen polarity, such as "*not only good but amazing*". Issues of word sense, similar to those discussed in section 2.2.2, present further problems. For example "the National *Trust* may waste money" conveys an opinion which expresses the polar opposite of trust. The domain of the sentiment being expressed can also effect polarity. "*Go read the book*" may be considered positive within a book review, however for a film it is generally see as negative.

At its heart sentiment classification poses a significant linguistic challenge, and the approaches vary as a result of it. They can be broadly split into two approaches however, supervised and unsupervised. Unsupervised methods propose that sentiment can be understood by analysing its linguistic form. By understanding the rules which allow sentiment to be expressed, we should be able to both identify and understand it within a sentence. Supervised methods suggest that the complexities of language make unsupervised methods too specific and difficult to identify. Instead it hopes to make use of machine learning's supervised techniques in order to better classify sentiment. We will explore and contrasts these two methods. In particular, we will focus upon the unsupervised approach put forward by Turney (3), and the supervised approach proposed by Pang et al. (2).

2.2.3.1 *Unsupervised sentiment classification*

Turney (3) suggests a two part approach to supervised sentiment classification. As discussed when exploring subjectivity in section 2.2.2, adjectives tend to be a significant grammatical structure through which sentiment is expressed. Thus, Turney proposes extracting phrases containing adjectives and whose structure indicates an expression of sentiment. Given a sentence, we tag its parts of speech, before extracting any two-word phrases whose structure can be found within the following linguistic patterns:

Table 2.1: Extraction patterns for identifying opinionated two-word phrases

Rule	First word	Second word	Third word (*not extracted*)
1.	JJ	NN, NNS	anything
2.	RB, RBR, RBS	JJ	not NN, not NNS
3.	JJ	JJ	not NN, not NNS
4.	NN, NNS	JJ	not NN, not NNS
5.	RB, RBR, RBS	VB, VBD, VBN, VBG	anything

Once these phrases have been identified, we can then determine their sentiment's polarity. This is done by first selecting two words commonly associated with strong positive and negative sentiment. Turney suggests *excellent* and *poor* as the benchmark words for positive and negative polarity. This is largely due to their prevalent use within reviews as descriptions for high and low ratings. In order to calculate a phrases sentiment, we attempt to measure the association between it and benchmark's words. Co-occurence between two words is calculated using their *Pointwise Mutual Information (PMI)*, defined as:

$$\mathrm{PMI}(word_1, word_2) = \log_2\left(\frac{\mathrm{p}(word_1 \ \& \ word_2)}{\mathrm{p}(word_1)\,\mathrm{p}(word_2)}\right) \tag{2.5}$$

Where $\mathrm{p}(word_1, word_2)$ is the probability that $word_1$ and $word_2$ co-occur within a corpora, and $\mathrm{p}(word)$ is the probability that $word$ occurs. Now that we have definition for PMI, we can define the *semantic orientation (SO)* of a $phrase$ as:

$$\mathrm{SO}(phrase) = \mathrm{PMI}(phrase, \text{"}excellent\text{"}) - \mathrm{PMI}(phrase, \text{"}poor\text{"}) \tag{2.6}$$

The resulting semantic orientation is a measure of a phrase's sentiment. An SO larger than 0 denotes positive polarity, while an SO less than zero indicates negative polarity. Thus, a sentence's overall polarity is simply the average of its phrases' SO. This approach to supervised sentiment classification has proven effective across a variety of review domains. Turney reports an impressive 80% when classifying bank reviews and an even better 84% accuracy for automobile reviews. He does note however, that movie reviews present a challenge for his supervised approach, reporting an accuracy of 65.83% within the movie domain. Nonetheless, across domains Turney reports classification rates of 74.39%, demonstrating the strong potential which lies within unsupervised methodologies.

2.2.3.2 *Supervised sentiment classification*

Shortly after Turney published his paper on supervised approaches (3), Pang et al. put forward a counter paper. This addressed the potential of supervised learning within the same domain of internet reviews as Turney's original paper. At its core, Pang et al. address the issue that often sentiment can be expressed in very subtle ways. For example, "*How could anyone sit through this movie?*" does not express negative opinion in any readily apparent way. Essentially the proposition put forward by Pang et al. is that the nuanced structures through which we express opinion are too vast and varied. They cannot simply be whittled down into a simple set of rules, and rather, we should look to experience to guide our classification.

As with any supervised problem, the learning experience is largely guided by our choice of features. Before we examine these, it is important to introduce the concept of *n-grams* and how they work as features. For example, if we use unigram feature set, there is a feature for every possible word. A feature set this large is unnecessary however, as the only words which will be important in classification are those we encounter in training. Thus we build a feature set from the words we encounter when training. If our training set only contained "*I love the NHS*", we would have the following feature set for classification $\langle f_I, f_{love}, f_{the}, f_{NHS} \rangle$. Alternatively if we used bigrams (2 word phrases), we would have a feature set $\langle f_{(I,love)}, f_{(love,the)}, f_{(the,NHS)} \rangle$. But what values do we assign to these features when given a sentence to classify? Pang et al. experiment with two options:

1. *Term presence* denotes whether the n-gram phrase that a feature represents occurs within our sentence. For example, using the unigram and bigram feature sets above, and given a sentence "*I hate the NHS*", we would have the following feature sets:

$$\langle f_I, f_{love}, f_{the}, f_{NHS} \rangle = \langle true, false, true, true \rangle$$
$$\langle f_{(I,love)}, f_{(love,the)}, f_{(the,NHS)} \rangle = \langle false, false, true \rangle$$

2. *Term frequency* denotes how frequently each feature's n-gram phrase occurs within our sentence. For example, using the unigram and bigram feature sets above, and given a sentence "*I hate the NHS, but I love my GP*", we would have the following feature sets:

$$\langle f_I, f_{love}, f_{the}, f_{NHS} \rangle = \langle 2, 1, 1, 1 \rangle$$
$$\langle f_{(I,love)}, f_{(love,the)}, f_{(the,NHS)} \rangle = \langle 1, 0, 1 \rangle$$

Pang et al. also experiment with appending POS tags to the end of each word, thus distinguishing between their possible uses. In order to handle negation, any words between a negative word such as *not* and the next punctuation mark are tagged with a NOT. For example "I do not like the NHS" would result in a feature set $\langle f_I, f_{do}, f_{not}, f_{NOT-like}, f_{NOT-the}, f_{NOT-NHS} \rangle$.

The different feature sets were tested within the movie review domain. The presence feature set for unigrams performs strongest in their experiments with an accuracy of 82.9%. The combination of unigrams and bigrams sees a marginal drop in accuracy to 82.7%. Interestingly POS tags also have a slight negative effect on accuracy, seeing it drop to 81.9% when coupled with a unigram presence feature set. In domains where the expression of sentiment is subtle, supervised approached have a clear benefit over their unsupervised counterparts. However, supervised learning requires one to build a training set, which can often prove time consuming. Furthermore its understanding of sentiment is based upon experience, thus it could never really explain why it reached its decision. Deciding which approach is better is difficult, and we shall explore this is more detail in section 7.

2.2.3.3 Present research and issues

Recent research has focussed on how combinations of supervised and unsupervised learning can be used to improve classification rates. Essentially these improvements have hoped to introduce greater linguistic detail into the supervised approach described by Pang et al.. In the following section we shall provide a general overview of two improved methodologies put forward by Wilson et al. (13) and Benamara et al. (14). We shall explore these approaches in greater detail in section 7.

Although Wilson et al. (13) acknowledge the need for elements of supervised learning, they observe that the sentence-level approach put forward by Pang et al. is to general. Instead they propose that to truly understand sentiment, we must approach it at a phrase level. The main motivation behind this is the common misclassification of *clue* words as polar, when the sense in which they are being used means they are in fact neutral. This problem is of particular relevance to the supervised approach discussed above. The method put forward by Pang et al. essentially creates a lexicon of polar words during training and later uses them as clue's for classifying polarity. As mentioned in our introduction to opinion classification, this can lead to words being taken out of context to classify neutral statements as polar. Wilson et al. propose a two step solution to this. The first step identifies all clue phrases within a sentence, before using a supervised approach to classify each one as polar or neutral. The polarity of each polar phrase is then disambiguated to give it an overall classification of either *positive*, *negative* or *both*. Not only does this approach provide a more rigorous framework for sentiment classification, unlike the methods put forward above it also acknowledges the potential neutrality of phrases within a sentence.

Alongside the influential research into phrase-level sentiment by Wilson et al., other prominent research has focussed on measuring sentiment strength. Benamara et al. (14) highlight the important role of adverbs as measures of opinion. These adverbs are known as *adverbs of degree*. Within this subset of adverbs, five clear classifications can be noted:

1. *Affirmation* adverbs such as *certainly* and *absolutely* strengthen adjectives.
2. *Doubt* adverbs such as *possibly* and *seemingly* weaken adjectives.
3. *Strong intensifying* adverbs such as *exceedingly* and *extremely* strengthen adjectives.
4. *Weak intensifying* adverbs such as *barely* and *scarcely* weaken adjectives.
5. *Negation/minimising* adverbs such as *hardly* and *rarely* invert or neutralise adjectives.

Using a lexicon containing adverbs of degree and their appropriate classification, all unary and binary adverb adjective combinations are found. A unary combination has the form $\langle adverb \rangle \langle adjective \rangle$, whilst a binary combination has the form $\langle adverb_i, adverb_j \rangle \langle adjective \rangle$. Each adjective in the matching phrases has its polarity strength adjusted according to the classification of the adverbs which proceed it. For unary combinations the score is a product of the adjective and adverb strengths. For binary combinations, the strength of $\langle adverb_j \rangle \langle adjective \rangle$ is calculated first as if it were a unary combination, before calculating the strength combination of the resulting score and $adverb_i$. Benamara et al. report results almost on par with human strength classification, highlighting the proposed method as not only viable but effective.

Although significant improvements have been made within the field, sentiment classification is still far from perfect. Many of its problems have been reduced in size, however they have not been eradicated. One could argue that this is largely due to the statistical nature of supervised learning, and clearly the field still has a lot to learn from linguistics. Most importantly to this project however, is the fact that little research has explored beyond the confines of polarity and into the realm of emotion. We shall explore the field's limited approach to the classification of emotion in more detail later, in section 2.4.

2.2.4 Topic extraction

2.3 Sentiment on Twitter

With the recent and rapid growth of Twitter has come an interest in understanding the sentiment expressed on it. Although at its heart an issue of sentiment analysis, Twitter's constraints and protocols pose new and different issues for current approaches. Literature is still limited, and solutions to the problems within it are varied. In this section we will focus on some of the more prominent approaches. In particular we will outline the framework proposed by Barbosa and Feng (15), whilst looking at some of the innovative improvements and observations put forward by Go et al. (16) and Bermingham et al. (17).

Barbosa and Feng (15) propose a two stage sentiment analysis framework. Firstly the subjectivity of a tweet is determined, and if subjective, the tweet's sentiment is then classified. This framework bares many similarities to sentence-level sentiment classification, however the approach within each stage is in many ways very different. Particular emphasis is placed upon the need for strong subjectivity detection. There is a lot of *noise* on Twitter through adverts and spam accounts, thus it is important to filter this out if we ever hope to obtain an accurate overview. A typical noisy tweet might be:

> *Get a FREE $500 Starbucks Gift Card >> Special Online Offer ...: Starbucks is celebrating its first forty years ... http://bit.ly/iepyV5*

Barbosa and Fang propose some previously unconsidered features for helping distinguish noise from subjective tweets. As evident from analysing the tweet above, Barbosa and Fang note that *link presence* and *uppercase letter frequency* serve as particularly useful subjectivity clues. A novel approach is taken to training the subjectivity classifier using existing online Twitter sentiment classifiers. Subjective tweets are scraped from three such sites, and any tweet appearing as subjective in all three is added to the training data. They report that although this can lead to slight bias, it serves as an effective bootstrapping method.

Barbosa and Fang take an entirely supervised approach to polarity classification using many of the features discussed in section 2.2.3. Uppercase letter frequency again proves particularly useful, along with a feature for *good emoticons*. An emoticon is a text-character face expressing an emotion, for example happy is commonly represented as `:)` while sad is `:(`. Barbosa and Fang note significant improvements both in subjectivity and sentiment classification when using tweet-based features, as opposed to the typical approaches described in sections 2.2.2 and 2.2.3. Using unigrams alone for sentiment classification, Barbosa and Fang report an error rate of 44.5%, whilst the introduction of Twitter based features reduces this to 25.1%. Although far from perfect, the improvements are notable, and suggest that a better understanding of the intricacies of Twitter could lead to further improvements.

Interestingly recent work by Bermingham and Smeaton (17) suggests that further linguistic detail when building a feature set in fact harms classification rates. Rather than using POS tagging or larger n-grams, they note that features such as link presence and punctuation mark frequency serve as far better discriminators for subjectivity and polarity. They report accuracy rates of 74.85%, which are strikingly similar to those achieved by Barbosa and Fang.

Building a training set for Twitter can prove difficult due to the need for large data sets. There is an extraordinary diversity of structure, language and grammatical approach on Twitter, thus a large training set is necessary if we hope to be able to accurately classify its broad range of opinion. Further to the innovative approach taken by Barbosa and Fang, Go et al. (16) suggest a further innovative technique for quickly building a large data set. By searching Twitter for all tweets containing positive and negative emoticons, Go et al. were able to quickly assemble list of polarised opinion. This method for building a training set proved remarkably successful, and simply using unigrams as features, they report an accuracy rate of 82.2%.

Although literature regarding sentiment analysis on Twitter is limited, there have been significant advances in accuracy. Interestingly, as Bermingham and Smeaton observe, detailed linguistic features seem to be of little benefit when classifying subjectivity and polarity. However, as noted by Go et al., the size of the training set seems to have a marked effect on classification accuracy. Clearly Twitter poses many new challenges for sentiment analysis, and although progress has been made, more in depth research is needed before the best approaches can be truly identified.

2.4 Emotion

Defining emotion has been a problem that has puzzled philosophers and thinkers as far back as Cicero and Descartes. Although there is no unifying theory, or completely accepted classification, in general many have agreed there to be two broad types of emotion, the *basic emotions* and *complex emotions*. Basic emotions are biologically innate within all humans, whilst complex emotions are culturally specific amalgamations of our basic one. Deciding upon both what constitutes are basic and complex emotions however has been the cause of significant debate. Perhaps the two most prominent classifications of recent times are those put forward by Ekman (18) and Plutchik (19).

2.4.1 Current defenitions

After years of work within the field, and having observed the Fore tribesmen of Papua New Guinea, Ekman's 1969 paper presented what he believed to be the six core emotions. These were *anger*, *disgust*, *fear*, *happiness*, *sadness*, *suprise*. Within his work he notes that the Fore tribesmen could identify these emotions when presented with photos of faces expressing them, regardless of their cultural origin.

In 1980, Robert Plutchik (19) presented his research into human emotion. Within it, he uses five of the emotions put forward by Ekman, whilst introducing three new emotions (see figure 2.1). Plutchik expands these emotions further, referring to them as *dimensions*, within which different emotions can express varying degrees of their dimension. Furthermore, each of the eight emotion definitions also has a polar opposite definition within the list.

Figure 2.1: Robert Plutchik's eight basic emotions *(* proposed by Ekman)*

Basic Emotion	Polar Emotion	Degrees (strong to weak)		
Joy	Sadness	Ecstasy	Joy	Serenity
Trust	Disgust	Admiration	Trust	Acceptance
Fear*	Anger	Terror	Fear	Apprehension
Surprise	Anticipation	Amazement	Surprise	Distraction
Sadness*	Joy	Grief	Sadness	Pensiveness
Disgust*	Trust	Loathing	Disgust	Boredom
Anger*	Fear	Rage	Anger	Annoyance
Anticipation*	Surprise	Vigilance	Anticipation	Interest

Taking this original list of eight, Plutchik also proposesa further eight complex emotions, formed from combinations of the original eight (see figure 2.2).

Figure 2.2: Robert Plutchik's eight complex emotions

Combined basic emotions	Complex Emotion	Polar Emotion
Anticipation *and* Joy	Optimism	Disappointment
Joy *and* Trust	Love	Remorse
Trust *and* Fear	Submission	Contempt
Fear *and* Surprise	Awe	Aggressiveness
Surprise *and* Sadness	Disappointment	Optimism
Sadness *and* Disgust	Remorse	Love
Disgust *and* Anger	Contempt	Submission
Anger *and* Anticipation	Aggressiveness	Awe

The level of detail within Plutchik's research provides a wider scope of definition than that put forward by Ekman. For this reason it shall serve as the classification system we attempt to computationally replicate. Furthermore, Plutchik's proposal introduces concepts of polarity and strength to emotion. Both these concepts bare strong similarities to research within sentiment classification 2.2.3, and we will explore the benefits of this similarity in chapter 7.

2.4.2 Computational classification

2.5 Tools and languages

Although speed is not the primary performance measure within this project, we still hope to maximise efficiency where possible. This desire to strike a balance between programming practicality and speed has defined our choice of tools and languages. As a result languages have primarily been chosen based upon their practicality with regards to rapid development and experimentation, On the other hand, data critical tools such as database software have been chosen with the hope of maximising efficiency whilst remaining flexible. The remainder of this section shall focus upon which languages and tools we have chosen and why, along with any potential pitfalls we may encounter in using them.

2.5.1 Ruby

Ruby (www.ruby-lang.org) is a high-level interpreted programming language. It was deemed well suited to the project for several reasons:

- Its functional aspects make data manipulation simple and fluid. This is particularly relevant for sentiment analysis in which fast data manipulation is essential, especially when building feature sets or partitioning data for training.
- Ruby is easily extensible with C/C++, and support for JAVA libraries is stable. This means that we are not simply limited to Ruby libraries when building our classifiers. As most research into sentiment analysis and NLP has been conducted in JAVA or C++, this extensibility is useful in allowing us to make use of leading work and libraries.
- Ruby's focus on simplicity means that components can be rapidly developed and experimented with. The freedom afforded by it's simplicity means that we can focus on experimenting with our approach rather than focussing on the complexities of lower-level languages such as C or C++.
- Ruby has well-established frameworks for building web-services such as *Sinatra* and *Ruby on Rails*. These allow database-driven websites to be rapidly developed and will enable both simple collection and visualisation of data.

This project draws upon some core Ruby frameworks, in particular making use of:

Sinatra is a simple web application framework for Ruby. In essence it allows methods to be easily bound to specific web addresses and is perfect for rapidly developing sites. It is stable and well supported by the open source community. Within this project it shall be used to deliver the data visualisation and provide a backend through which tweets can be found and labelled in order to train our classifiers.

Artificial Intelligence for Ruby is a detailed library providing an array of machine learning focussed tools. In particular the library includes a Naive-Bayes classifier which will be used within each of our three classifiers.

LIBSVM is a well establish C++ library for support vector machines. This can be used within Ruby through a LIBSVM interface written by Tom Zeng[4]. As

[4]https://github.com/tomz/libsvm-ruby-swig

LIBSVM is written in C++ it is much faster than Ruby alternatives. Additionally it supports multi-class classification without any additional work.

2.5.2 MongoDB

MongoDB (www.mongodb.org) is a no-SQL document-oriented database. Essentially, MongoDB discards schemas and relationships, instead leaving us with the complete freedom to structure and organise documents as we choose. Documents (or objects) are stored using *JSON*[5] notation, and can be organised into *collections* of documents. For examples a collection statuses, might consist of a set tweets each stored within MongoDB as JSON objects, such as that in listing 2.1.

Listing 2.1: Example tweet stored within MongoDB

```
{
  "_id" : ObjectId("4dee4f6697dee90936000082"),
  "text" : "@CalebHowe It wasn't Weiner! http://flic.kr/p/9Rgb84
      #weinergate",
  "source" : "twitter_search",
  "source_id" : NumberLong("78133750344597504"),
  "created_at" : "Tue Jun 07 2011 17:18:46 GMT+0100 (BST)",
  "from_user" : "speciallist",
  "from_user_id" : 15966313,
  "to_user" : "CalebHowe",
  "to_user_id" : 8574053
}
```

As MongoDB is schema-less, the database makes no structural checks on existing data nor does it check when inserting new data. This means that if there are discrepancies such as additional fields, the database will neither complain or alert the user.

The simplicity of document-oriented databases has led to a rapid adoption amongst data-driven companies(20; 21), preferring versatility and speed over the benefits of relational databases. With this has come increasing research and stability, ensuring that databases such as MongoDB are not only fast and efficient, but stable and safe. MongoDB was selected for this project in particular, for several reasons:

- MongoDB's schema-less no-SQL approach is well suited to changes in our data's structure. Importantly, this means that if Twitter change their API's object model, adaptation is only required within code, rather than in the database as well. Furthermore, additional micro-blogs with new or different properties can be easily introduced and stored within the database.

[5] *JavaScript Object Notation is an open standard for defining data structures and objects. Although based upon JavaScript, methods for converting to and from JSON are supported in most languages.*

- Due to the lack of a relational layer, when inserting data and performing basic queries, MongoDB performs much faster than SQL derivatives(22). This is important, both if we hope to retrieve and classify large swathes of data from Twitter, and if we intend to train our classifiers using a large body of data.
- Relationships are of little importance to this project, and thus the benefits of a schema-less database far outweigh the negatives.

2.5.3 Processing.js

PART II | IMPLEMENTATION

3
OVERVIEW

Within the literature discussed, the approaches taken to sentiment analysis are varied. Each proposes its own unique takes on structure, components and design patterns. This project hopes to both utilise and draw together the more successful methods taken when analysing sentiment. Furthermore, in combining them we hope to discover potential improvements and avenues for further exploration. Lastly the project hopes to look at potential methods for further expanding sentiment analysis in order to explore a wider depth and range of emotion.

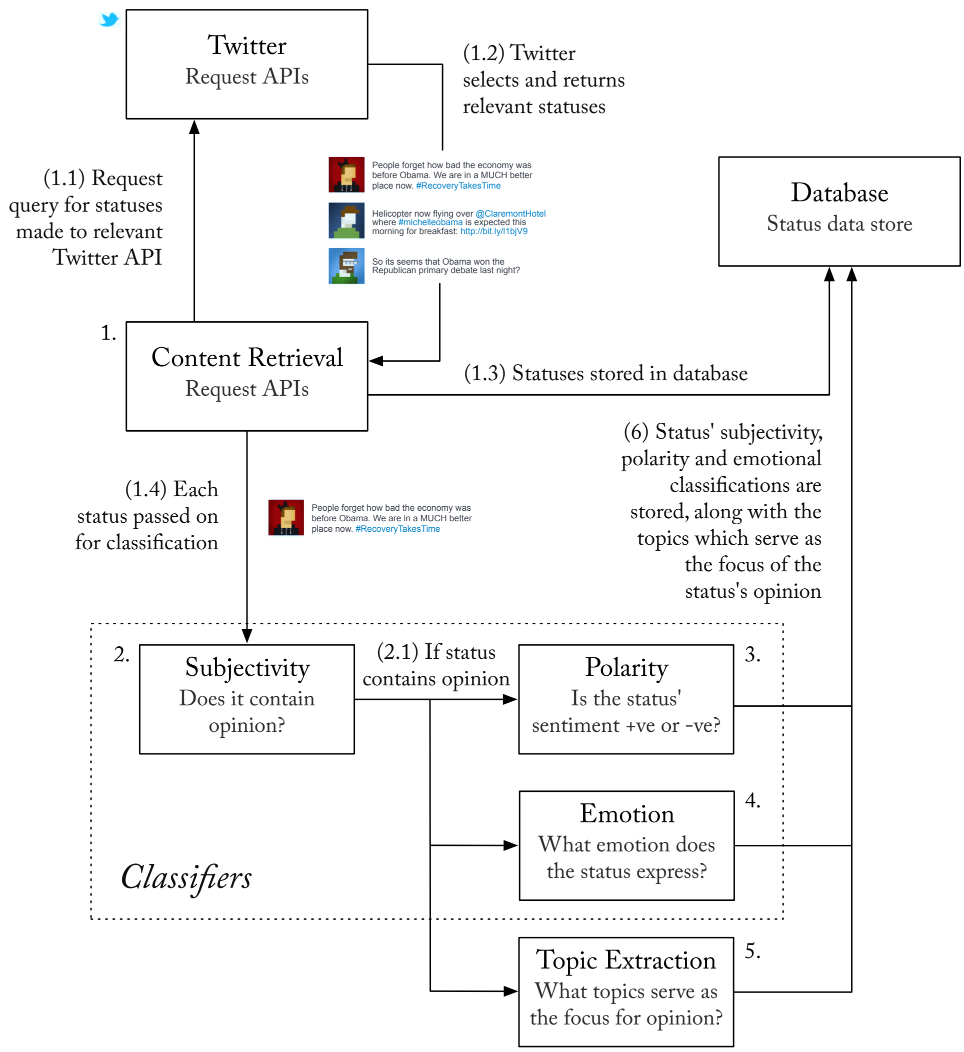
3.1 Design

The approach we will take within this project can be broadly divided into six core components:

1. *Content retrieval* - interfaces with Twitter's various APIs to retrieve relevant twitter statuses for classification. This will serve as the system's main input.
2. *Subjectivity classification* - given a twitter status, this module will serve as a mechanism for identifying whether the status is opinionated or not.
3. *Polarity classification* - given an opinionated status, this module hopes to classify it's polarity, along with the strength of the opinion expressed.
4. *Topic extraction* - given an opinionated status, this module will determine the topic's at which the opinion is directed.
5. *Emotion classification* - given an opinionated status, this module hopes to determine the emotional state of the tweet, classified according to the labels put forward by Plutchik.
6. *Classification storage* - once classified, statuses must be persisted for use within external modules, such as visualisation. The storage of the necessary data will be handled by this module.

The way in which these components interlink is best demonstrated visually, as shown in figure 3.1.

Figure 3.1: Structure for sentiment analysis



We will discuss the first five components in more detail in their own relevant chapters. Classification storage will be referenced where relevant in other chapters, however it does not warrant detailed discussion on its own.

3.2 Implementation narrative

Through our discussion of the project's implementation we shall use five example tweets to help illustrate the classification process within the system.

No.	Text
1.	I think David Cameron is doing a rather good job: strong leader, holding together seemingly impossible coalition, keeping labour at bay
2.	BBC Obama, Merkel warn on Europe debt http://bbc.in/jfZW3I
3.	#Obama says European debt crisis can't be allowed to threaten global economy. Shame US never felt this way about its financial crisis.

4

CONTENT RETRIEVAL

In order to analyse sentiment on Twitter, the first problem posed is how exactly we retrieve the necessary data for analysis. In essence there are two types of data we need, Twitter data for labelling in order to train our system and a much larger set of Twitter data to classify in order to better understand sentiment on Twitter. We will draw these two slightly disparate elements together under the banner of content retrieval, as although their intended use is quite different, they share many similarities in the way they are collected and stored. The remainder of this chapter shall examine the Twitter APIs relevant to this project and how they are used, before

4.1 Data structure and storage

As described above, information regarding statuses[1] serves two purposes within this project, firstly when labelled it can serve as training data, and if unlabelled, it can be classified in the hope of better understanding the overall sentiment on Twitter. Essentially this means that every status retrieved can be expanded upon with additional information pertaining to both an annotator's and-or classifier's decisions as to its sentiment labels. Although MongoDB does not require that documents added to it adhere to any schema, the requirements of our project insist that the basic attributes illustrated in listing 4.1 are present. Both `trained_status` and `classified_status` are optional.

Listing 4.1: Basic JSON structure for status objects stored in MongoDB

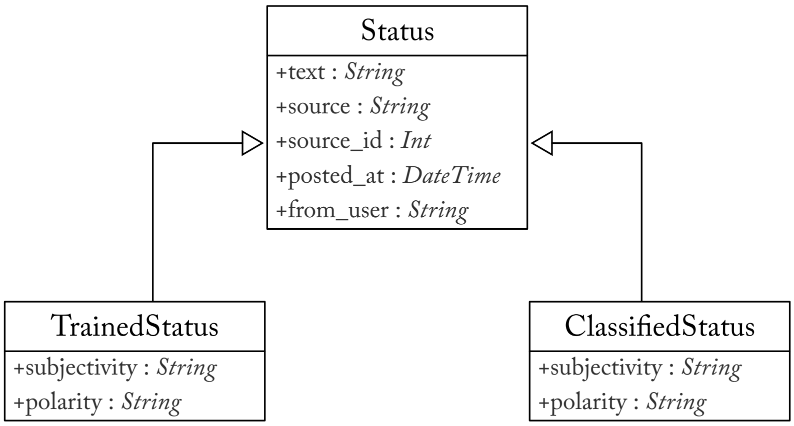
```
{
  "text" : "Hi, this is an example tweet!"
  "source" : "search_api" | "streaming_api"
  "source_id" : NumberLong("78133750344597504"),
```

[1]We will use the terms *status* and *tweet* interchangeably throughout this project to refer to the same concept. Not only do Twitter use the term status within their object model, but it is also semantically more accurate for this project which eventually hopes to classify statuses from a variety of micro-blogging services.

```
  "posted_at" : "Tue Jun 07 2011 17:18:46 GMT+0100 (BST)",
  "from_user" : "joeroot"
  "trained_status" : {
    "subjectivity" : "subjective" | "objective" | "spam"
    "polarity" : "positive" | "negative" | "neutral"
  }
  "classified_status" : {
    "subjectivity" : "subjective" | "objective" | "spam"
    "polarity" : "positive" | "negative" | "neutral"
  }
}
```

In order to express this JSON structure within Ruby, we will create three new classes, *Status*, *TrainedStatus* and *ClassifiedStatus*. The core status class will consist of all the attributes described within our JSON schema (listing 4.1), along with attributes for retrieving a *Status*'s relevant training or classification data. Both the *TrainedStatus* and *ClassifiedStatus* classes will maintain the *Status* class' original attributes, along with new ones for accessing any relevant label data. As shown in figure A.2, both *TrainedStatus* and *ClassifiedStatus* inherit attributes from their parent *Status* class.

Figure 4.1: Class structure for representing tweets/statuses



As we go on to explore other components in later chapters, we will add and expand upon this initial core set of attributes and methods.

The simplicity of Mongo means any additional data which might be discovered regarding tweets can be inserted as new attributes without any schema worries. In order to map our MongoDB data to Ruby classes we have used an open-source library, *MongoMapper*. MongoMapper gives us the ability to map class objects and attributes to documents within MongoDB. This means that rather than loading and storing an object's attributes in memory, MongoMapper will always read and write directly to the relevant database document. Listing B.1 demonstrates how this mapping is setup for the *Status* class.

4.2 Retrieving content on Twitter

Twitter offers three core APIs for accessing their data, all of which return the said data in either JSON or ATOM format. The three APIs offered are:

1. *REST API* - Twitter's REST API provides direct access to Twitter's data model. Requests can be made to access a wide variety of Twitter data, such as user information, timelines, statuses and trends. Essentially Twitter makes all of their sites functionality available through it's API.
2. *Search API* - Twitter's search API makes their internal status search engine available to developers. Given a query and set of parameters, the search API will find all relevant statuses and return them in the requested format. The status objects returned contain less information than those in the REST API.
3. *Streaming API* - Twitter's streaming API grants developers access to a high-throughput near-realtime percentage of Twitter's live data. In general twitter makes 1% to 10% of all statuses available through the API and includes further methods for filtering by keywords.

This project shall make use of the *search* and *streaming* APIs. The search API will be used to find tweets for labelling whilst the streaming API shall be employed to collect and filter data for classification. As noted in section 4.1, data is stored in MongoDB as JSON objects, thus all API calls will request that results be returned in JSON.

4.2.1 Search API

Twitter's search API allows developers to search for tweets matching a specified set of criteria. This is made available through a web-based API, accessible by making GET requests to `http://search.twitter.com/search.json`. Parameters can be passed to the API through the request headers. For example appending `q=david-%20cameron` to the request headers would prompt the API to search for all statuses containing the words "david" and "cameron". Twitter formats the returned data as JSON, with the search results being stored as an array of tweets in the `results` variable, as demonstrated in listing B.2.

In order to search and store Twitter data, we created a singleton *TwitterSearch* class. It's `search` method takes a query string and a parameters hash[2]. The parameters hash is used to pass in optional arguments, which map directly to Twitter's search API's optional arguments. Optional parameters include restricting tweets to

[2]ruby parameters

certain languages through the `locale` parameter or the `until` parameter for finding tweets up to a specified date. The query string and parameters are then combined to generate a request URL. Once generated, the `open-uri` library is used to make a GET request to the URL, and the returned JSON data is stored in a local variable for formatting. Each result's JSON data is amended with a `source` attribute to identify that the status was retrieved using the Twitter search API. The remaining JSON data however is left unchanged and intact. Once amended, each result is initialised as a *Status* object using it's JSON data. When initialised, MongoMapper automatically writes the data to MongoDB for later use, thus all results are persisted within our database.

Although there are plenty of Ruby libraries available for accessing Twitter's search API, they are often bloated. Many implement their own class structure for representing Twitter's object model which is both unhelpful and unnecessary for this project. Furthermore, although some offer cacheing, none provide support for database persistence. Instead our choice of MongoDB and MongoMapper mean that simply through retrieving the JSON results and using it to initialise *Status* objects, we will have persistent access to the results.

4.2.2 Streaming API

Twitter's graded streaming APIs deliver developers realtime Twitter data. The three grades, *spritzer*, *garden-hose* and *fire-hose* offer different volumes, ranging from 1% to 100% of all realtime tweets. This project shall utilise the *garden-hose* level, which delivers between 1% and 10% of all live statuses. Furthermore Twitter offer API methods for filtering the stream ensuring that only tweets matching an array of keywords are included. Twitter make this available by opening an HTTP connection, but never closing it. Tweets are then passed down this connection to the developer. The method call is made to `http://stream.twitter.com/1/statuses-/filter.json` using the `track` parameter to pass in the set of keywords. For example appending `track=nhs,david%20cameron` to the request header will filter the stream for all tweets containing "nhs" or "david cameron". The results are formatted as JSON with each result being separated by a line break. The returned data is slightly more detailed than that returned by the search API, however the core data still adheres to the same schema.

4.3 Pre-processing

Pre-processing hopes to determine the linguistic attributes of a status which have not already been computationally identified. Primarily we are interested in identifying any grammatical meta-data, in this case each word's part of speech tag, along with any Twitter meta-data the word might contain such as hashtags and mentions. Although not necessarily features themselves, this additional detail will help better describe the data and thus prove useful when building features which truly understand the linguistic nature of Twitter statuses.

4.3.1 Part of speech tagger

Numerous approaches have been taken to part-of-speech tagging, however for the purpose of this project we will implement a Ruby adaptation of Coburn's Perl part of speech tagger (http://search.cpan.org/ acoburn/Lingua-EN-Tagger). Coburn's approach is dependant upon a large corpora of text, within which each word has been annotated with its part of speech tag. For each word within the corpora, we calculate the probability of it being used as a certain part of speech, given by how often it occurs as that said part of speech. For example, if the word *like* appears twice as a an infinitive verb, and once as a past tense verb, the probability of it being an infinitive verb will be $\frac{2}{3}$, and the probability of it being a past tense verb will be $\frac{1}{3}$.

$$\Pr(tag \mid word) = \frac{|\,\text{occurences}(word \text{ as } tag)|}{|\,\text{occurences}(word)|} \tag{4.1}$$

Once this is done, we then use the corpora to calculate the probability of a tag occurring, given the previous word's tag. For example, if an infinitive verb is followed by a proper noun twice and an adjective once, then the probability of a word being a proper noun given the word before it was an infinitive verb is $\frac{2}{3}$, and the probability of it being an adjective is $\frac{1}{3}$.

$$\Pr(tag \mid tag_p) = \frac{|\,\text{occurences}(tag_p \text{ preceeding } tag)|}{|\,\text{occurences}(tag_p)|} \tag{4.2}$$

Thus, when determining a word's part of speech tag, we want to find the tag which maximises the product of the two probabilities, given our current word and the part of speech tag for the word which preceded it.

$$\text{pos}(word, tag_p) = \underset{tag \in tags}{\arg\max}(\Pr(tag \mid word) \cdot \Pr(tag \mid tag_p)) \tag{4.3}$$

Our *TweetTagger* class replicates the above behaviour in Ruby. Colburn's pretrained probabilities are stored in two separate text files and read into two hashmaps when a *TweetTagger* object is instantiated. In order to tag a *String* object with it's appropriate part of speech tags, we can call the `fetch_tags(text)` method, passing in the text we wish to tag as a parameter. Before we tag the text, it is first split on it's white space and punctuation, using our `split(text)` function, as demonstrated in listing 4.2.

Listing 4.2: Example use of split function

```
TweetTagger.new.split("Hello, please split this string.")
  => ["Hello", ",", "please", "split", "this", "string", "."]
```

Once split, we are now able to tag each word and punctuation mark with it's appropriate part of speech tag. This is done through the `assign_tag(word, previous_tag)` method, a Ruby implementation of equation 4.3. Where a word does not exists within our lexicon of trained probabilities it is referred to the `unknown_word(word)` method which attempts to find the word elsewhere. This method will be of particular use when adapting the tagger for Twitter in section 4.3.2.

Once every word has been tagged, the `fetch_tags(text)` method returns a JSON formatted, ordered array of words and their corresponding tags. For example, if we were to call `fetch_tags` with *Example 1*, the returned JSON would be as shown in listing 4.3.

Listing 4.3: Returned part of speech tags for *Example 1*

```
TweetTagger.new.fetch_tags("I think David Cameron is doing a
    rather good job: strong leader, holding together seemingly
    impossible coalition, keeping labour at bay")
  =>  [{"word" : "I", "tag" : "prp"}, {"word" : "think", "tag" :
     "vbp"}, {"word" : "David", "tag" : "nnp"}, {"word" : "
     Cameron", "tag" : "nnp"}, {"word" : "is", "tag" : "vbz"}, {
     "word" : "doing", "tag" : "vbg"}, {"word" : "a", "tag" : "
     det"}, {"word" : "rather", "tag" : "rb"}, {"word" : "good",
      "tag" : "jj"}, {"word" : "job", "tag" : "nn"}, {"word" : "
     :", "tag" : "pps"}, {"word" : "strong", "tag" : "jj"}, {"
     word" : "leader", "tag" : "nn"}, {"word" : ",", "tag" : "
     ppc"}, {"word" : "holding", "tag" : "vbg"}, {"word" : "
     together", "tag" : "rb"}, {"word" : "seemingly", "tag" : "
     rb"}, {"word" : "impossible", "tag" : "jj"}, {"word" : "
     coalition", "tag" : "nn"}, {"word" : ",", "tag" : "ppc"}, {
     "word" : "keeping", "tag" : "vbg"}, {"word" : "labour", "
     tag" : "nn"}, {"word" : "at", "tag" : "in"}, {"word" : "bay
     ", "tag" : "nn"}]
```

Our implementation of Colburn's part of speech tagger proves both fast and accurate on spell-checked bodies of text. However, it performs less well when confronted with the abbreviations, acronyms and mis-spellings common amongst Twitter statuses. Furthermore, it fails to account for Twitter protocols such as hashtags and mentions. It is these issues we will confront in our next section, 4.3.2.

4.3.2 Tagging Twitter statuses

This section shall examine the improvements and additions we have made to Colburn's original tagger in order to better suit it to the task of tagging Twitter statuses.

4.3.2.1 URLs

URLs are frequently used within tweets, however Colburn's original design provides no mechanism for handling or tagging them. In order to account for this we decided to introduce an additional *URL* tag to the Penn tag-set along with amending our original tagger to both identify and tag URLs.

But how does this fit into out *TweetTagger* class? As discussed in the previous section, any words which cannot be found in the lexicon are passed on to the *unknown_word(word)* method. Typically this is used to pick up words representing ordinal and float numbers which are not included in the word-probability lexicon. If the unknown word is not a number, other identifiers are examined, such as it's suffix, in the hope that they might provide additional clues for identifying the word's POS. For example words suffixed with "*ly*" tend to be adverbs or adjectives, thus if the actual word cannot be found in the word-probability lexicon, we assign it tag properties based upon all words ending in "*ly*". In order to match unknown words to their most appropriate identity, such as a number or suffixed word, each word is compared against a regular expression[3] which corresponds to each potential identity in a series of *if-else* statements. Statements higher up the list take priority, thus for identities which provide certainty such as numbers are placed first. In order to help better illustrate this, example code from the method itself in included in listing 4.4.

Listing 4.4: Example if-else statements for handling unknown words

```
def classify_unknown_word(word)
  if /-?(?:\d+(?:\.\d*)?|\.\d+)\z/ =~ word
    classified = "*NUM*"  # Floating point number
  elsif /\A\d+[\d\/:-]+\d\z/ =~ word
    classified = "*NUM*"  # Other number constructs
  elsif /\A-?\d+\w+\z/o =~ word
    classified = "*ORD*"  # Ordinal number
  elsif
    ...
  end
  return classified
end
```

[3] Explain regular expressions

In order to determine whether a word is in fact a URL, an additional condition is added to check whether the unknown word matches against our URL regular expression (see table A.2). With the tagger now amended, if a tweet containing a URL is passed in, the URL is split off into it's own word and tagged as a URL, as shown in listing 4.5. Finally it is important to note that as we have now introduced a URL tag, we need to introduce a probability set for our new tag into the tag-probability lexicon. This is used for calculating the most appropriate tags for any word which directly follow URLs. URLs are typically used as nouns when not at the end of sentences, thus rather than retrain our tagger with a small corpora of tweets, we chose to simply assign the same following-tag probabilities to URLs as we have for nouns.

Listing 4.5: Example use of split function

```
TweetTagger.new.fetch_tags("BBC Obama, Merkel warn on Europe debt
    http://bbc.in/jfZW3I")
  =>[{"word" : "BBC", "tag" : "nnp"}, {"word" : "Obama", "tag" :
     "nnp"}, {"word" : ",", "tag" : "ppc"}, {"word" : "Merkel",
     "tag" : "nnp"}, {"word" : "warn", "tag" : "vbp"}, {"word" :
      "on", "tag" : "in"}, {"word" : "Europe", "tag" : "nnp"}, {
     "word" : "debt", "tag" : "nn"}, {"word" : "http://bbc.in/
     jfZW3I", "tag" : "url"}]
```

4.3.2.2 *Mentions*

As with URLs, mentions are not handled by our initial implementation of Colburn's tagger. As mentions represent a unique entity, rather than needing a new tag, they are in fact proper nouns. Essentially they are used as replacements for names, thus this is both a natural and correct assumption to make. Although we decided to tag mentions as proper nouns, we also wanted to design a way of acknowledging that the word also contains Twitter meta-data. In order to do this we decided to expand upon our initial JSON representation to include a *meta* object for each word. Within this meta object we are then able to include additional Twitter data, and in the case of mentions, this is done with a boolean *mention* tag to denote whether the word is being used as a Twitter mention, as in listing 4.6.

Listing 4.6: Example JSON structure for representing a mention word

```
{
  "word" : "@BBC",
  "tag" : "nnp",
  "meta" : {"mention" : true}
}
```

Are approach to identifying mentions is similar to that used for URLs. As no words beginning with "@" exist within our lexicon, it is safe to assume they will be passed on to the `unknown_word(word)` method. Within here an additional condition is added for identifying mentions. The word is then tagged as a proper noun, and it's meta object's mention attribute is set to true.

4.3.2.3 *Hashtags*

As with mentions, although hashtags are not directly supported by our tagger implementation, they do not require their own part of speech tag, and instead if processed correctly can often be used as natural parts of speech. Often users will amend keyword within their statuses with hashtags, without interrupting the flow of the status, as seen in *example 3* with the opening word, "#Obama". Furthermore hashtags are often used to express opinion, for example "#hate" is an extremely popular hashtag for expressing intense dislike for the status' target. Evidently, understanding hashtags genuine part of speech is therefore important.

The *TweetTagger* class approaches this by first setting hashtag words' meta object's boolean *hashtag* attribute to true and it's *original* attribute to the hashtag word. Once this is done the hashtag is stripped of it's opening hash, and the word is classified as per usual. For example, in the case of "#Obama", we would see the corresponding representation in listing 4.7.

Listing 4.7: Example JSON structure for representing a hashtag word

```
{
  "word" : "Obama",
  "tag" : "nnp",
  "meta" : {
    "mention" : false
    "hashtag" : true
    "original" : "#Obama"
  }
}
```

As with both URLs and mentions, this is achieved by adding an additional condition to the `unknown_word(word)` method. Unlike with mentions and URLs however, the word is stripped of it's opening hashtag, and passed back for retagging, thus finding it's genuine part of speech tag within the context of it's position in the status text.

4.3.2.4 *Names, misspelling and acronyms*

4.4 Labelling data

The primary use of our search API interface, *TwitterSearch*, is to ease the process of collecting data for labelling. Rather than doing this within code, it was deemed simpler and quicker to build a web-based interface for searching and classifying tweets. The interface is built using a combination of HTML, CSS and JavaScript for the front-end, whilst the back-end data management is handled by a Ruby web-framework, Sinatra.

5

CLASSIFIERS

At the heart of this project's sentiment analysis lie our three classifiers for subjectivity, polarity and emotion. For reasons we shall discuss in their respective chapters, we have elected to take a supervised approach to learning for all three classifiers. As a result, our classifiers' implementations share much much in common, and this chapter shall look at how this commonality was unified through our *Classifier* class. This project is largely experimental in nature, looking at how we can best adapt, use and develop existing and new ideas for classifying sentiment on Twitter. As a result, we wanted to design a parent *Classifier* class which would best remove the complexities of correctly assembling and training our classifiers. Instead we wanted a class which would allow us to focus on experimenting with different feature sets and classification techniques, along with providing tools for gathering and comparing our results. The remainder of this chapter shall first outline our core aims for the class, before going on to outline our approach. After this we will summarise the methods discussed and implemented within the class, before finally evaluating it's performance against our original aims.

5.1 Overview

As discussed above, our *Classifier* class hopes to unify the overall approach taken to classification across the three classifiers. In doing so it hopes not only to save time by eradicating repetition, but also simplify and encourage experimentation through intelligent design. The three core aims of the class are:

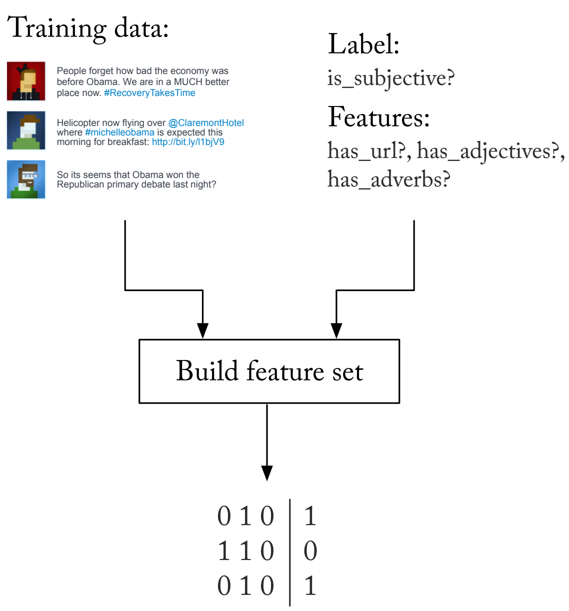
1. *Simple feature selection* when initialising a classifier. For example, we might want to initialise a classifier with just *feature one* before initialising another classifier for a performance comparison, with both *feature one* and *feature two*.
2. *Simple classifier method selection* when initialising a classifier. For example we may want to compare the performance of our classifiers when using a Support Vector Machine against using a Naive Bayes classifier.
3. *Unified testing* for classifiers in which key machine learning metrics such as accuracy, recall, precision and f-measure are automatically calculated. This

will ease in comparing and contrasting results for changes in feature set and classification method.

5.2 Preparing the data

In order to train our classifier, we first need to prepare our *training data* accordingly. This means building a feature set for our training data, according to both the *features* we want to use, and the *label* we with which we are looking to classify statuses, as shown in figure 5.1.

Figure 5.1: Outline for building feature set



We wanted our approach to this to be as generic as possible, and in doing so made use of Ruby's `send` method. Every object within Ruby has a `send` method which accepts a symbol[1] and a series of arguments. When called, Ruby will in turn attempt to call the object's method which corresponds to the given symbol, along with any additional arguments. For example `"Hello, Joe".send(:split, ", ")` is in effect the same as `"Hello, Joe".split(", ")` and will return `["Hello", "Joe"]`. It is this we make use of when trying to take a flexible approach to feature choice. It is important to note that all feature methods take a *Status* object and are object level methods within the class extending our *Classifier* class. Thus, using the send method, we can initialise our classifier with an array of symbols representing feature methods, before using this array to generically create our training set, as shown in listing 5.1.

[1] explain ruby symbol

Listing 5.1: Method for building a status's feature set, where `@features = [:feature_1,..., :feature_n]`

```ruby
def build_status_feature_set(status)
  return @features.map{|f| self.send(f, status)}
end
```

Thus given a training set, we simply iterate over it, creating a feature set for each training status. This array of feature sets can now be used to train our classifier.

Before we train our classifier, we also need the an array containing each of the training statuses appropriate label, such as the status' polarity or subjectivity. It is this label which are classifier will be trained to identify, and just as we defined an array of features to build our feature sets, we will instantiate a single `label` variable used to store the label's method symbol, such as `:is_subjective?` or `polarity`. This is built in a similar manner to our feature sets, using the object `send` method coupled with the `label` symbol, and is wrapped up in the `fetch_status_label(status)` method.

5.3 Training and classifying

With our training data now prepared in the form of two arrays, one of feature sets and one of labels, we can now train our classifier. As explored by Pang et al. (23), the classification method used can impact the effectiveness of our results. We decided to experiment with the two leading probabilistic and non-probabilistic methods, Naive Bayes classifier and Support Vector Machines.

Rather than building our own SVM and NB classifiers, we elected to use two well established libraries, LIBSVM and AI4R, as discussed in section 2.5. As both utilise their own methods and input schemas for training and classification, we implement two intermediary interface for both the LIBSVM and the AI4R libraries. Both our *NaiveBayes* and *SupportVectorMachine* classes serve as this layer, and share the same public methods. Effectively this allows us to initialise both our classifiers using the feature sets and labels generated above, as demonstrated in listing 5.2.

Listing 5.2: Example initialisation of LIBSVM and AI4R Naive Bayes classifiers through intermediary layer

```ruby
feature_sets = training_statuses.map{|s| self.
    build_status_feature_set s}
labels = training_statuses.map{|s| self.fetch_status_label s}
svm = SupportVectorMachine.new(feature_sets, labels)
nb = NaiveBayes.new(feature_sets, labels)
```

Whenever a class extending our *Classifier* class is initialised, an additional parameter is passed in alongside the features and label, denoting whether to use a SVM or NB classification method. A new classification method is initialised as described above, and stored within the *Classifier* object's `classifier` variable. Once the *NaiveBayes* and *SupportVectorMachine* layers have been initialised and trained, they can be used to classify statuses through their `classify(feature_set)` method. When given a feature set, this method will classify it and return the appropriate label. This enables us to again make use of the `build_status_feature_set(status)` method from within our *Classifier* class, rather than having to generate a feature set in the appropriate format for the specified classifier method. In order to better illustrate the chain of command, we have included the *Classifier* class' `classify(status)` method in listing 5.3.

Listing 5.3: Classifier class' classify method

```
def classify(status)
  feature_set = self.build_status_feature_set status
  # @classifier is either an initialised SupportVectorMachine or
  # NaiveBayes object. This is instantiated when initialising our
  # Classifier object.
  return @classifier.classify(feature_set)
end
```

With unobtrusive support for different feature sets and classification methods now implemented within our *Classifier* class, we can finally look at the approach to performance testing.

5.4 Testing

Due to the project's strong element of experimentation, providing a robust framework for evaluating our classifiers was essential. As is common within both machine learning and sentiment analysis, our four chosen measure of performance are *accuracy*, *precision*, *recall* and *f-measure*. These four metrics combined give a fairly clear portrait of a classifiers strengths and weaknesses. In order to better explain each measure we shall first introduce four terms commonly used within binary classification.

True positives are the set of of correctly classified documents for the positive label. For example in subjectivity classification, this could be taken to be all statuses correctly classified as subjective.

True negatives are the set of correctly classified documents for the negative label. For example in subjectivity classification, this could be taken to be all statuses correctly classified as not subjective.

False positives are the set of incorrectly classified documents, who have been labelled positive when they are in fact negative.

False negatives are the set of incorrectly classified documents, who have been labelled negative when they are in fact positive.

Using these four definitions, we can now go onto better define our performance measures:

Accuracy is used to measure how many documents have been correctly classified across the entire training set.

$$accuracy = \frac{|TP \cup TN|}{|TP \cup FP \cup TN \cup FN|} \tag{5.1}$$

Precision is a measure of how accurate our positively labelled data actually is. This is done by looking at what fraction of positively labelled data is actually positive.

$$precision = \frac{|TP|}{|TP \cup FP|} \tag{5.2}$$

Recall is a measure of how much of our positive data is correctly labelled by the classifier. This is done by looking at what fraction of positive data was correctly labelled.

$$recall = \frac{|TP|}{|TP \cup FN|} \tag{5.3}$$

F-measure combines the classifiers precision and recall rates to give an overall measure of accuracy.

$$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{5.4}$$

With the definitions for our measures in place, we now had to introduce a suitable method for making use of them. The first method, `test(k,statuses)`, performs *k-fold cross validation* across the labelled statuses passed in to the method. Effectively k-fold cross validation divides our labelled statuses up into k partitions, before iterating through the them, using $(k-1)$ partitions as training data, and 1 partition as testing data. This is repeated k times, and the average measures across each of the folds are taken as the overall test's measures.

Although this proved suitable for binary classification, it is not suitable for multiclass classification. Instead for classification with two or more output labels, the *precision*, *recall* and *f-measure* are returned for each potential label. As *accuracy* measures correct classification across all labels, there was no need for it's results to be calculated for each possible label. The full implementation details for this can be seen in listing ??.

We felt however that for strenuous testing one test was not enough, and instead a `repeat_test(k,statuses,n)` method was introduced. This repeats the `test(k,statuses)` method n times, before returning the average of the measures across those n repetitions.

5.5 Class summary

Before we evaluate the performance of our classifier, we will give a quick overview of the core method names along with a description of their inputs, outputs and purpose.

Method	Arguments	Returns	Description
`initialize`	`features`, `label`, `classifier_method`, `statuses`	`none`	Intialisation method for creating new Classifier objects. Will intialize and train either a SVM or NB classifier, with the specified `features` and `label`, using the labelled `statuses` as training data.
`classify`	`status`	`label`	The classify method takes a `status` object, converts it into the appropriate feature set before using the trained classifier method to classify the feature set and return the appropriate `label`.
`test`	`k`, `statuses`	`accuracy`, `precision`, `recall`, `f-measure`	Given a set of `statuses`, will split the data into `k` folds, before testing and training with each. Returns average `accuracy`, `precision`, `recall` and `f-measure` across the `k` folds.
`repeat_test`	`k`, `statuses`, `n`	`accuracy`, `precision`, `recall`, `f-measure`	Runs `test(k, statuses)` `n` times, returning the average `accuracy`, `precision`, `recall` and `f-measure` across those `n` repetitions.

5.6 Evaluation

The *Classifier* class proved a fundamental class and tool within this project. It allowed for the remainder of the implementations focus to be placed upon innovative feature use and performance, rather than a struggle for consistency across different classifiers. In order to better evaluate the classifier, we shall examine to what extent it met it's original three aims:

1. The approach taken to simple feature switching meant that new ideas could be experimented and developed easily, without having to worry about adapting our code elsewhere.
2. As with feature selection, the class' ability to swap classification methods was simple and effective. It required no additional work throughout the remainder of the project and made evaluation much simpler.
3. The strong performance testing suite provided a very simple method for meaningfully comparing features and classification methods. This proved vital in understanding how best to approach feature and method selection, along with being fundamental to our evaluation process.

Although no numeric measure can be given to express the class' performance, the fact that it met each of its original aims and the freedom it gave to focus on innovation and testing clearly marks it a success.

6
SUBJECTIVITY CLASSIFICATION

Once a status has been retrieved, the first step towards understanding it's sentiment lies in determining whether it is subjective or objective. This is known as subjectivity classification and it serves as the first stage within our sentiment analysis engine. In classifying subjectivity, we decided to take a supervised approach to the problem. This is largely due to the fact, that as Wiebe and Riloff observe (11), although unsupervised approaches' precision rates are high, achieving high recall rates is remarkably difficult. As a result, the problem now lies in designing a supervised classifier which can best separate statuses into their correct labels.

In this chapter we shall first examine how we built our training set, and the reasoning behind our labelling. With a training set in place, we will then go on to explore the features we think will be of value, along with the reasoning behind them and a discussion of their implementation. We shall then go on to look at the results of our tests, examining which feature combinations performed best and why, along with which classification method best suited the problem. Finally we shall evaluate our classifiers performance against the results commonly seen in literature.

6.1 Training set

Before we can build a classifier, we need to assemble a training set which best represents the domain of our problem. Although we have already discussed our approach to labelling data in chapter 4, we shall examine how this relates to subjectivity classification.

6.2 Features

As with any classification problem, picking a suitable feature set is decisive in classification performance. In implementing our classifier we chose to draw upon a range of previously successful features, along with new or adapted ones of our own. In this section we shall examine why and how we implemented our chosen features, but will save a discussion of their effectiveness and our final choice until later, in this chapters's results and evaluation sections. Much of the work covered by our pre-processing part of speech tagger is utilised within our feature selection.

6.2.1 Adjectives

As discussed in the background section, adjectives are often regarded as strong indicators of subjectivity. With our status' POS tags readily available through its `parts_of_speech` method. Within our tagset however, there are four different tags representing adjectives. In order to handle this, we added a `general(pos)` method to our *TweetTagger* class. This method takes specific tag, such as "*jjr*" or "*adr*", and returns its more general tag such as "*adj*" or "*adverb*". This is done by comparing the specific tag against regular expressions corresponding to our more general tags. These more general tags, their meaning and their regular expression can be seen in table A.3.

With a method now available for identifying tags as adjectives, we needed a way of collecting any words being used as adjectives within our statuses. As this a method corresponding to an attribute of our status, we decided to implement this `adjectives` method within our *Status* class, as in listing 6.1.

Listing 6.1: Status object method for returning all words being used as adjectives within the status

```
def adjectives
  self.parts_of_speech.select{|pos| TweetTagger.general(pos["tag"
     ]) == "adj"}
end
```

It is important to note that this design decision to keep methods such as adjective collection within *Status* objects is consistent throughout our implementation. Table ?? contains a list of *Status* methods, along with their return values and a short description of their purpose.

With a status' adjectives now easily accessible we were able to build our two adjective-based feature methods:

`has_adjectives?` returns a boolean value denoting adjective presence within the status.

`no_adjectives` returns one of three values based upon the number of adjectives. For zero adjectives, `0` is returned, for one or two adjectives, `1` is returned and for three or more adjectives `2` is returned.

6.2.2 URLs

Our approach to URLs was similar to that taken for adjectives. Using a status' `urls` method, we build our two url-based feature methods:

`has_urls?` returns a boolean value denoting URL presence within the status.

`no_urls` returns one of three values based upon the number of URLs. For zero URLs, `0` is returned, for one or two URLs, `1` is returned and for three or more URLs `2` is returned.

6.2.3 Subjective clues

As originally observed by Wiebe and Riloff (9), subjective clues often prove to be effective discriminators when classifying subjectivity. In effect, this is done by compiling a list of clue words, alongside their subjectivity strength, in our case weak or strong. Furthermore the word's part of speech tag is noted, so as to ensure that the word being marked as a clue is in fact being used in the correct sense.

Our approach to clue finding uses the same lexicon as Wiebe and Riloff (9). Alongside this we use out own lexicon of subjective words, as collected in section 6.1. The clue data is stored is stored in regular text files, with each line consisting of one clue, as in listing 6.2.

Listing 6.2: Example clue from the subjective clue lexicon

```
type=weaksubj len=1 word1=block pos1=noun stemmed1=n
    priorpolarity=negative
type=weaksubj len=1 word1=block pos1=verb stemmed1=y
    priorpolarity=negative
```

The `type` field represents whether a clue is *strong* or *weak*, the `len` field denotes the length of the clue. The `word`, `pos` and `stemmed` fields represent the properties of each word in the clues phrase, with `stemmed` meaning whether the clue applies to all un-stemmed versions of the word. For example, this means that not only is "*block*" a clue in the above example, but so is the word "*blocks*" when it is used as a verb. Finally the *priorpolarity* field denotes the polarity of the clue.

In order to find our clues, we implemented a singleton *ClueFinder* class. The class loads each clue into a `clues` hashmap, in which each *key* is a clue phrase, and it's *value* is an array of all possible ways in which the phrase may be used as a subjective clue, as demonstrated in listing 6.3.

Listing 6.3: Ruby hashmap representation of listing 6.2

```ruby
clues["block"]
  => {[
```

```
    {:type => "weaksubj", :len => 1, :pos => ["noun"], :stemmed
        => ["n"], :priorpolarity => "negative"},
    {:type => "weaksubj", :len => 1, :pos => ["verb"], :stemmed
        => ["y"], :priorpolarity => "negative"}
  ]}
```

With our clues now loaded in a hashmap, we defined a `clue_data(words, pos)` method which when given a phrase an array of words along with its corresponding POS tags, will check the hashmap to see if the combination does in fact represent a clue. If they do, the method will return the clue `type` and `priorpolarity`, otherwise it will simply return `nil`. Using this we could now easily define three useful methods for our *Status* class, `subjective_clues`, `weak_subjective_clues`, `strong_subjective_clues`. These methods simply iterate over the statuses unigrams, bigrams and trigrams each time checking to see if they represent a clue, before filtering them accordingly if we are looking for weak or strong clues.

With status methods now in place for easily retrieving clues, we can go on to build our six clue-based feature methods.

`has_subjective_clues?` returns a boolean value denoting the presence of one or more subjective clues

`no_subjective_clues` returns one of three values based upon the number of subjective clues. For zero clues, `0` is returned, for one or two clues, `1` is returned and for three or more clues `2` is returned.

`has_weak_subjective_clues?` as with `has_subjective_clues?`, but only noting weak clues.

`has_strong_subjective_clues?` as with `no_subjective_clues`, but only noting weak clues.

`no_weak_subjective_clues` as with `has_subjective_clues?`, but only noting strong clues.

`no_strong_subjective_clues` as with `no_subjective_clues`, but only noting strong clues.

6.2.4 Capitalised words

As observed by Barbosa and Fang (15), often subjective links to articles or in some worst case scenarios spam, tend to contain significant capitalisation. We experimented with two features based upon this.

`capitalised_word_frequency` looks at how the ratio of capitalised words to total word, i.e.

$$c.w.f = \frac{|words_{capitalised}|}{|words|} \tag{6.1}$$

Rather than returning the floating point number, one of three values are returned. For all values between 0 and 0.3, we return `0`, for values between 0.3 and 0.5, we return `1` and for values greater than 0.5, we return `2`.

`capital_letter_frequency` looks at the ratio of capitalised letters to total letters, i.e.

$$c.l.f = \frac{|letters_{capitalised}|}{|letters|} \tag{6.2}$$

Rather than returning the floating point number, one of three values are returned. For all values between 0 and 0.2, we return `0`, for values between 0.2 and 0.5, we return `1` and for values greater than 0.5, we return `2`.

6.3 Results

6.4 Evaluation

7 POLARITY CLASSIFICATION

"The PM's 18 week waiting time pledge will not mean any change for how the #NHS is operating: http://bit.ly/kSjpfL" interesting, how do we know no change is good?

7.1 Training set

7.2 Features

7.3 Results

7.4 Evaluation

8

EMOTION CLASSIFICATION

9

TOPIC EXTRACTION

10

DELIVERY

PART III | EVALUATION

11 EVALUATION

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

12

CONCLUSION

BIBLIOGRAPHY

[1] B. Liu, ``Sentiment analysis and subjectivity,'' *Handbook of Natural Language Processing,*, 2010.

[2] B. Pang and L. Lee, ``Thumbs up?: sentiment classification using machine learning techniques,'' … *of the ACL-02 conference on* …, 2002.

[3] P. Turney, ``Thumbs up or thumbs down?: semantic orientation applied to unsupervised classification of reviews,'' *Proceedings of the 40th Annual Meeting on* …, 2002.

[4] P. Norvig, ``On chomsky and the two cultures of statistical learning,'' May 2011.

[5] B. Pang, ``A sentimental education: Sentiment analysis using subjectivity summarization based on minimum cuts,'' 2004.

[6] R. Mihalcea and C. Banea, ``Learning multilingual subjective language via cross-lingual projections,'' … *MEETING-ASSOCIATION FOR* …, 2007.

[7] J. M. Wiebe, R. F. Bruce, and T. P. O'Hara, ``Development and Use of a Gold-Standard Data Set for Subjectivity Classifications,'' in *the 37th annual meeting of the Association for Computational Linguistics*, (Morristown, NJ, USA), pp. 246--253, Association for Computational Linguistics, 1999.

[8] J. Wiebe, ``Effects of adjective orientation and gradability on sentence subjectivity,'' *Proceedings of the 18th conference* …, 2000.

[9] J. Wiebe, ``Learning subjective adjectives from corpora,'' *Proceedings of the National Conference on Artificial* …, 2000.

[10] J. Wiebe, ``Subjectivity word sense disambiguation,'' *Proceedings of the 2009* …, 2009.

[11] J. Wiebe, ``Learning extraction patterns for subjective expressions,'' in *Proceedings of the 2003 conference on Empirical* …, 2003.

[12] J. Wiebe, ``Word sense and subjectivity,'' … *of the 21st International Conference on* …, 2006.

[13] J. Wiebe, ``Recognizing contextual polarity in phrase-level sentiment analysis,'' *Proceedings of the conference on* …, 2005.

[14] F. Benamara and C. Cesarano, ``Sentiment analysis: Adjectives and adverbs are better than adjectives alone,'' *Proceedings of the* …, 2007.

[15] L. Barbosa, ``Robust Sentiment Detection on Twitter from Biased and Noisy Data,'' *research.att.com*.

[16] A. Go and R. Bhayani, ``Twitter sentiment classification using distant supervision,'' *CS224N Project Report*, 2009.

[17] A. Bermingham, ``Classifying sentiment in microblogs: is brevity an advantage?,'' *Proceedings of the 19th ACM ...*, 2010.

[18] P. Ekman, ``The repertoire of nonverbal behavior: Categories, origins, usage, and coding,'' *Semiotica*, 1969.

[19] R. Plutchik, ``The nature of emotions,'' *American Scientist*, 2001.

[20] A. Popescu, ``Cassandra @ twitter: An interview with ryan king,'' February 2010.

[21] J. Moore, ``Big data @ foursquare,'' March 2011.

[22] M. Kennedy, ``Mongodb vs. sql server 2008 performance showdown,'' April 2010.

[23] B. Pang, ``Opinion mining and sentiment analysis,'' *Foundations and Trends in Information Retrieval*, 2008.

[24] M. Marcus and M. Marcinkiewicz, ``Building a large annotated corpus of English: The Penn Treebank,'' *Computational ...*, 1993.

PART IV | APPENDIX

A

TABLES AND FIGURES

A.1 Background

Table A.1: The University of Pennsylvania (Penn) tagset, as proposed by Marcus et al. (24)

Tag	Part of speech	Example
CC	Conjunction, coordinating	and, or
CD	Adjective, cardinal number	3, fifteen
DET	Determiner	this, each, some
EX	Pronoun, existential there	there
FW	Foreign words	
IN	Preposition / Conjunction	for, of, although, that
JJ	Adjective	happy, bad
JJR	Adjective, comparative	happier, worse
JJS	Adjective, superlative	happiest, worst
LS	Symbol, list item	A, A.
MD	Verb, modal	can, could, 'll
NN	Noun	aircraft, data
NNP	Noun, proper	London, Michael
NNPS	Noun, proper, plural	Australians, Methodists
NNS	Noun, plural	women, books
PDT	Determiner, prequalifier	quite, all, half
POS	Possessive	s, '
PRP	Determiner, possessive second	mine, yours
PRPS	Determiner, possessive	their, your
RB	Adverb	often, not, very, here
RBR	Adverb, comparative	faster
RBS	Adverb, superlative	fastest
RP	Adverb, particle	up, off, out
SYM	Symbol	
TO	Preposition	to
UH	Interjection	oh, yes, mmm

VB	Verb, infinitive	take, live
VBD	Verb, past tense	took, lived
VBG	Verb, gerund	taking, living
VBN	Verb, past/passive participle	taken, lived
VBP	Verb, base present form	take, live
VBZ	Verb, present 3SG -s form	takes, lives
WDT	Determiner, question	which, whatever
WP	Pronoun, question	who, whoever
WPS	Determiner, possessive	question, whose
WRB	Adverb, question	when, how, however
PP	Punctuation, sentence ender	., !, ?
PPC	Punctuation, comma	,
PPD	Punctuation, dollar sign	$
PPL	Punctuation, quotation mark left	
PPR	Punctuation, quotation mark right	"
PPS	Punctuation, colon, semicolon, elipsis	:, ..., -
LRB	Punctuation, left bracket	(, {, [
RRB	Punctuation, right bracket	), },]

A.2 Content retrieval

Figure A.1: Class structure for representing tweets/statuses

Figure A.2: Class structure for representing tweets/statuses

Table A.2: Regular expressions for matching features

<table>
<tr><th>Feature</th><th>Regular expression</th></tr>
<tr><td>URLs</td><td><code>/(?:http|https):\/\/[a-z0-9]+(?:[\-\.]{1}[a-z0-9]+)\-</code>
<code>.[a-z]{2,5}(?:(?::[0-9]{1,5})?\/[\s])?/ix</code></td></tr>
</table>

Table A.3: Regular expressions for generalising part of speech tags

General term	General tag	Regular expression
Adjective	`adj`	`/jj[rs]*/`
Noun	`noun`	`/nn[sp]*/`
Verb	`verb`	`/vb[dgnpz]*/`
Adverb	`adverb`	`/r((b[rs]*)|p)/`
Pronoun	`pronoun`	`/(ex)|(wp)/`

B

CODE EXAMPLES

B.1 Content retrieval

Listing B.1: Illustration of Status class' MongoMapper attributes

```
class Status
  include MongoMapper::Document

  # Attributes
  key :text, String
  key :source, String
  key :source_id, Int
  key :posted_at, DateTime
  key :from, String

  # Relationship attributes
  key :classified_status, ClassifiedStatus
  key :trained_status, TrainedStatus
end
```

Listing B.2: Example Twitter search API results


```
{
  "results":[
    {
      "text":"@twitterapi, look at my example tweet!",
      "to_user_id":396524,
      "to_user":"TwitterAPI",
      "from_user":"jkoum",
      "metadata":
      {
        "result_type":"popular",
        "recent_retweets": 100
```

```
      },
      "id":1478555574,
      "from_user_id":1833773,
      "iso_language_code":"nl",
      "profile_image_url":"http://twitter.com/image.jpg",
      "created_at":"Wed, 08 Apr 2009 19:22:10 +0000"
    }
  ],
  "since_id":0,
  "max_id":1480307926,
  "refresh_url":"?since_id=1480307926&q=%40twitterapi",
  "results_per_page":15,
  "next_page":"?page=2&max_id=1480307926&q=%40twitterapi",
  "completed_in":0.031704,
  "page":1,
  "query":"%40twitterapi"
}
```