

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

I would like to thank my parents, girlfriend Elizabeth, Tim (sorry!) and Catherine for all their support over the past few months. I can't tell you how much of a difference each one of you has made. I'd like to thank Susan Eisenbach for her continual advice, support and open door over the past three years. Thanks also to my second marker, Iian Phillips, for his willingness to help and general input to the project.

Lastly, I would like to thank my supervisor Francesca Toni. Your insightful input, enthusiasm and encouragement have kept me going on the numerous occasions I've felt like giving up. Thank you for taking on my project, its (sp?) genuinely been a pleasure!

ABSTRACT

In a day and age where public opinion dictates so much within politics and business, finding better ways to understand and harness it are proving increasingly fundamental. This project will look at how we can use today's online forums of debate such as Twitter, as a source for more accurate methods of understanding public emotion and sentiment.

In doing so we will explore techniques from Machine Learning and Natural Language Processing, along with the more specific emerging field of Sentiment Analysis. We will look at ways of both identifying opinion and classifying its polarity. Whilst doing this, we shall look at the limitations of common methods within the field, along with identifying potential improvements and adaptations to them. Finally we shall look at how using Twitter as a domain will impact our approach to sentiment analysis, along with how its problems, and thus methods, must change as a result of Twitter's constraints and peculiarities.

Lastly, we shall look at how we can best draw our work together in order to make it available for others to use and understand. We will explore in which it can be made easily accessible to developers, along with methods for presenting it in an intuitive and understandable manner to others.

CONTENTS

I	Analysis	11
1	Introduction	13
1.1	Motivation	13
1.2	Contributions	14
2	Background	15
2.1	Twitter	15
2.2	Sentiment analysis	16
2.3	Supervised learning	17
2.4	Discovering opinion	20
2.5	Classifying opinion	23
2.6	Topic extraction	27
2.7	Sentiment on Twitter	28
2.8	Emotion	30
2.9	Tools and languages	32
II	Implementation	37
3	Overview	39
3.1	Design	39
3.2	Implementation narrative	40
4	Content retrieval	43
4.1	Data structure and storage	43
4.2	Retrieving content on Twitter	45
4.3	Pre-processing	47
4.4	Labelling data	52
4.5	Evaluation	53
5	Classifiers	55
5.1	Overview	55
5.2	Preparing the data	56
5.3	Training and classifying	57
5.4	Testing	58
5.5	Class summary	60
5.6	Evaluation	61

6	Subjectivity classification	63
6.1	Training set	63
6.2	Features	64
6.3	Results	67
6.4	Evaluation	72
7	Polarity classification	75
7.1	Training set	75
7.2	Features	76
7.3	Results	79
7.4	Evaluation	84
8	Emotion classification	87
8.1	WordNet	87
8.2	Building an emotion lexicon	89
8.3	Classifying emotion	91
8.4	Evaluation	92
9	Topic extraction	93
9.1	Extracting patterns	93
9.2	Identifying patterns	94
9.3	Evaluation	94
10	Delivery	95
10.1	Sentiment analysis engine	95
10.2	Online API	96
10.3	Visualisation	96
10.4	Evalutation	97
III	Evaluation	99
11	Evaluation	101
11.1	Core aims	101
11.2	Additional aims	102
11.3	Summary	102
12	Conclusion	105
12.1	Contributions	105
12.2	Further work	105
IV	Appendix	111
A	Tables and figures	113

A.1	Background	113
A.2	Content retrieval	114
B	Code examples	117
B.1	Content retrieval	117
B.2	Polarity classification	118
B.3	Topic extraction	119

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. As a result, finding ways of gathering and understanding this opinion has increasingly proved 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 informative, 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 be used neither to find opinion on breaking news, nor across a large number of topics. Nonetheless, as methods for measuring public opinion have improved both in accuracy and detail, politicians and policy makers have increasingly turned to them not only for affirmation of their policies, but for guidance and new initiatives.

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

Based upon this, our project hopes to deliver the following core set of contributions,

- An engine for collecting and understanding sentiment on Twitter:
 - A Twitter scraper which will fetch live statuses from Twitter.
 - A method for determining whether fetched statuses are opinionated or not.
 - A method for identifying whether fetched statuses' opinion is positive or negative.
 - A method for identifying the core topics which our fetched statuses are discussing.
 - A means of persisting our statuses and the results of the above methods, such that they can be easily retrieved at any time.
- A web service which can serve as an interface to our engine, through which,
 - Requests can be made to classify a status, upon which the service should return the classification engine's results.
 - A method for requesting any classified data our engine may have stored.

Combined, these should provide a tool for collecting, analysing and storing sentiment from Twitter. We should be able to read and access this in order to gain a better understanding of overall sentiment on Twitter. The web service should make our data available to others so they can use it however they deem fit.

Additionally, if possible we hope to implement the following extensions to our project,

- Extend our sentiment analysis engine to give a more detailed account of a broader spectrum of emotion.
- Web based visualisation tool for better understanding and exploring sentiment on Twitter

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 2.4 - 2.6 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 2.8 we will explore emotion in general, particularly looking at its scope and ways of classifying it. Finally in section 2.9 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 uprisings 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,

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

²This 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 how sentiment is expressed within language, the field instead takes a statistical, and in his opinion, uninformed approach. This means that rather than determining sentiment by understanding the constructs which have formed it, the field uses a limited linguistic understanding to predict sentiment based upon past examples. 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.

2.3 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 in discovering opinion and determining polarity. Thus, the remainder of this section will discuss supervised learning with respect to the classification of sentences.

2.3.1 Defining the problem

In both discovering opinion and determining polarity we want to find an approximate hypothesis function h , for an actual function c , where c would perfectly perform either task. 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 $O = \{positive, negative\}$, such that:

$$h \approx c : S \rightarrow O \quad (2.1)$$

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, \dots, f_n for our sentences, such that:

$$h \approx c : \langle f_1, f_2, \dots, f_n \rangle \rightarrow O \quad (2.2)$$

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, i.e. $h = c$, 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:

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

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.

2.3.2 Naive Bayes Classifier

The *Naive Bayes (NB)* classifier is a probabilistic supervised classifier (6). Using Bayes' theorem, it statistically predicts an input's most likely classification based

upon previous experience. Thus given an input represented as the feature set, f_1, f_2, \dots, f_n , a Naive Bayes classifier will essentially classify it with label $o \in O$, according to:

$$\arg \max_{o \in O} \Pr(o|f_1, f_2, \dots, f_n) \quad (2.4)$$

In order to be able to predict this probability, the classifier learns from its training data, which is also represented as a series of feature sets.

Within sentiment analysis literature (7; 1), Naive Bayes classifiers are often experimented with, and unlike in more general applications (8), it often performs strongly. Naive Bayes classifiers also have the additional advantage of requiring only a limited amount of training data in order to be able to perform well.

2.3.3 Support Vector Machines

Unlike Naive Bayes classifiers, *Support Vector Machines (SVM)* are non-probabilistic, binary classifiers. Given a set of training data presented as feature sets, f_1, f_2, \dots, f_n , SVMs hope to find a hypothesis function which best divides the training data into its two respective classes (9). In essence, this is done by plotting the features in n dimensions, before trying to find the hyperplane which "best" separates the features into their respective classes. In finding the "best" hyperplane, SVMs look for the hyperplane with the largest n -dimensional distance between it and its nearest training example.

As Support Vector Machines present themselves as binary-classifiers, they need to be adapted in order to be able to classify more than two labels. This is referred to as multi-class classification. As outlined by Hsu et al., there have been numerous attempts at implementing multi-class classification using SVMs, with each presenting its own benefits. As Caruana et al. observe, typically SVMs outperform NB classifiers, and within sentiment analysis (7; 1) in particular, Support Vector Machines often perform strongest.

2.3.4 Un-supervised learning

Un-supervised learning is a fairly general term, often used to describe approaches to learning in which there is no labelled data to draw upon. Typically this means that data is instead clustered based upon attributes known or unknown, to the programmer. As a result of no core underlying approaches, un-supervised learning is difficult to define. However, within this project it shall refer to approaches which use no training data, and in particular those which apply no supervised techniques.

2.4 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. (10) 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 alike 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 more commonly used features, as discussed by Wiebe et al. (11):

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. (11) 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. (11) 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 gradability 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. (12), the presence of polarised, gradable adjectives is a strong measure of subjectivity and a useful feature.

Wiebe et al. (11) observed that using the first three features, coupled with a fea-

ture noting cardinal numbers presence, 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:

$$\begin{array}{ccccccc} & \text{verb} & & \text{noun} & & \text{pron.} & \text{pron.} \\ & \text{likes} & \text{big} & \text{snakes} & \text{but} & \text{I} & \text{hate} & \text{them.} \\ \text{She} & & & & & & & \\ \text{pron.} & & \text{adj.} & & \text{conj.} & & \text{verb} & \end{array} \quad (2.5)$$

2.4.1 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 WordNet³. 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.

2.4.2 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 (13) suggests a supervised approach using a small set of a hundred or so seed words, such as *good* and *bad*, 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

³WordNet is a detailed dictionary with additional levels of detail describing the semantic inter-linking between words. It will be used throughout this project and shall be discussed in more detail in section 8.1.

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. (14) describe a supervised method for bootstrapping an initial training set. A high precision, low recall rule based classifier, as originally proposed by Wiebe and Riloff (15), 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.

2.4.3 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. (16), and further refined by Akkaya et al. (14). 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. (14) 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.

2.5 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 2.8. 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 2.4, 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 seen 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 contrast 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).

2.5.1 Unsupervised sentiment classification

Turney (3) suggests a two part approach to supervised sentiment classification. As discussed when exploring subjectivity in section 2.4, 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 (<i>not extracted</i>)
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-occurrence between two words is calculated using their *Pointwise Mutual Information (PMI)*, defined as:

$$\text{PMI}(\text{word}_1, \text{word}_2) = \log_2 \left(\frac{p(\text{word}_1 \ \& \ \text{word}_2)}{p(\text{word}_1) p(\text{word}_2)} \right) \quad (2.6)$$

Where $p(\text{word}_1, \text{word}_2)$ is the probability that word_1 and word_2 co-occur within a corpora, and $p(\text{word})$ is the probability that word occurs. Now that we have definition for PMI, we can define the *semantic orientation (SO)* of a *phrase* as:

$$\text{SO}(\text{phrase}) = \text{PMI}(\text{phrase}, \text{"excellent"}) - \text{PMI}(\text{phrase}, \text{"poor"}) \quad (2.7)$$

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.

2.5.2 Supervised sentiment classification

Shortly after Turney published his paper on supervised approaches (3), Pang et al. (2) 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:

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

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:

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

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 approaches 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 in more detail in section 7.

2.5.3 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. (17) and Benamara et al. (18). We shall explore these approaches in greater detail in section 7.

Although Wilson et al. (17) acknowledge the need for elements of supervised learning, they observe that the sentence-level approach put forward by Pang et al. is too 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. (18) 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 $\langle \text{adverb} \rangle \langle \text{adjective} \rangle$, whilst a binary combination has the form $\langle \text{adverb}_i, \text{adverb}_j \rangle \langle \text{adjective} \rangle$. 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 $\langle \text{adverb}_j \rangle \langle \text{adjective} \rangle$ 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 2.8.

2.6 Topic extraction

Unlike subjectivity and polarity, research into topic extraction is not as well developed or scoped. As a result, the methods for topic extraction tend to be less unified in approach.

Somasundaran et al. (19) propose that a sentence's topics can be discovered through role labelling. Essentially this takes words such as "*fear*", and assigns the words around it an argument role, based upon a known set of rules. For example

fear has two arguments, so in "*we fear death*", argument one is said to be "*we*" whilst argument two is "*death*". Rules such as these are built for a core set of words, so that in any instance the arguments which these roles take can be identified to serve as topics.

Other research such as that proposed by Nigam et al. (20) suggests taking a purely supervised approach, in which unigrams are used to indicate whether a word is subjective based upon a prior labelled set. This however is dependant upon a large corpora of labelled data, and furthermore does not make any attempt to understand the linguistic patterns behind how a topic can be identified.

Clearly research is varied, and no concrete work can be offered as an example of a "typical" approach. In many ways this is largely due to the fact that what constitutes a topic is largely dependant upon who is looking for it and why. For example, for politicians looking to understand responses to policy change, the topics they want to identify may be emotional responses whilst for a company examining reviews it might be aspects of their product.

2.7 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 (21), whilst looking at some of the innovative improvements and observations put forward by Go et al. (22) and Bermingham et al. (23).

Barbosa and Feng (21) 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 bears 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 2.5. 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 2.4 and 2.5. 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 (23) 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. (22) 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 a 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.

2.8 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 what constitutes both 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 (24) and Plutchik (25).

2.8.1 Current definitions

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, surprise*. 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 (25) 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 proposes a 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 <i>and</i> Joy	Optimism	Disappointment
Joy <i>and</i> Trust	Love	Remorse
Trust <i>and</i> Fear	Submission	Contempt
Fear <i>and</i> Surprise	Awe	Aggressiveness
Surprise <i>and</i> Sadness	Disappointment	Optimism
Sadness <i>and</i> Disgust	Remorse	Love
Disgust <i>and</i> Anger	Contempt	Submission
Anger <i>and</i> 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 2.5, and we will explore the benefits of this similarity in chapter 7.

2.8.2 Computational classification

As with defining emotion, approaches to classifying it are broad and in general, differ widely. Each present their own definitions of emotion, and as a result approaches can vary quite widely. Although essentially a machine learning problem, unlike polarity classification for example, emotion classification can not be attributed to one single field. It is commonly researched within Natural Language Processing, however other fields such as Human-computer interaction, also present their own approaches.

As with most machine learning problems, classifying emotion can be approached both through supervised and un-supervised techniques. Yang et al. (26) propose a largely unsupervised approach to classification, largely based upon an emotion corpora built with distant supervision. In a similar fashion to Go et al. (22), blog posts are scraped for sentences containing emoticons. These sentences are then classified with an emotion based upon that which Yang et al. perceive the emoticon to hold. Once done, each word in any sentence containing an emotion is given a point-wise mutual information score, based upon how often it co-occurs with its sentence's emotion. With a corpora of words and their relation to any given emotion now built, Yang et al. classify new sentences by examining their corpora to see if any of the sentence's words exist within it. If any words do exist within their corpora, the sentence's score for the word's emotion is increased. Yang et. al see considerable success with this method, however the diversity of the emotion they can label is limited to measures of *valence* and *energy*. Furthermore, the approach

uses no manual annotation when testing, instead relying on its unsupervised labels, and thus it could be seen as merely favouring its own error.

In an alternative approach, Alm et al. (27) put forward a supervised method for classifying emotion. In training their classifier, Alm et al. manually annotate a corpus of 185 children's stories with one of six emotions, *angry*, *disgusted*, *fearful*, *happy*, *sad* and *surprised*. Interestingly, they note that their annotators only achieved an agreement rate of between 45 and 65%, suggesting that classification rates such as those seen in subjectivity and sentiment analysis can never truly be achieved. With their corpora now annotated, they draw upon thirty core features for training and classifying. Many of these features are similar to those used in polarity classification such as unigrams and adjectives. However, of particular interest and novelty is their use of WordNet as a feature for exploring how the semantic grouping of a word will effect a sentence's emotional classification. This is done by grouping WordNet words based upon their relation to each of the six emotions. Although Alm et al. do not report the success rates seen in work by Yang et al., their classifier performs well, on average achieving accuracy rates of 63% when determining whether emotion exists. They are less successful in classifying the actual emotion and are only able to achieve an F-score of 32% when classifying negative emotions such as fear, and 13% for positive emotions such as happiness.

It is clear from the work of Alm et al. (27) that classifying emotion with any degree of detail or certainty is a remarkably difficult process, and one in which there is a lot of progress still to be made. Although emotion classification across a limited range has seen some success, such as that seen by Yang et al., as one's definition of emotion increases in scope, our ability to accurately classify it drastically falls.

2.9 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 will focus upon which languages and tools we have chosen and why, along with any potential pitfalls we may encounter in using them.

2.9.1 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*⁴ 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

```
1 {
2   "_id" : ObjectId("4dee4f6697dee90936000082"),
3   "text" : "@CalebHowe It wasn't Weiner! http://flic.kr/p/9Rgb84 #
           weinergate",
4   "source" : "twitter_search",
5   "source_id" : NumberLong("78133750344597504"),
6   "created_at" : "Tue Jun 07 2011 17:18:46 GMT+0100 (BST)",
7   "from_user" : "speciallist",
8   "from_user_id" : 15966313,
9   "to_user" : "CalebHowe",
10  "to_user_id" : 8574053
11 }
```

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(28; 29), 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.
- Due to the lack of a relational layer, when inserting data and performing basic queries, MongoDB performs much faster than SQL derivatives(30). 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.

⁴*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.*

2.9.2 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 its 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 established C++ library for support vector machines. Originally proposed and built by Chang et al. (31), it has become a popular tool for SVM classification. This will be used within Ruby through a LIBSVM interface written by Tom Zeng⁵. As LIBSVM is written in C++ it is much faster than Ruby alternatives. Additionally it supports multi-class classification without any additional work.

MongoMapper is a Ruby framework for interfacing between MongoDB and Ruby. It provides an array of functionality, and in particular simplifies the task of

⁵<https://github.com/tomz/libsvm-ruby-swig>

building Ruby classes which map to MongoDB documents. Rather than fetching data from MongoDB and storing it in a Ruby object, MongoMapper creates an open link between the two. Thus whenever attributes are read or written to a Ruby object, they are in fact directly read or written to the corresponding MongoDB document.

2.9.3 HTML5 and the canvas

In delivering a visualisation of our classifiers results, we want to ensure that it can be viewed across as many devices as possible. As a result, we will use HTML5 and its canvas element. Unlike previous iterations of the HTML canvas, the canvas included in HTML5 offers a far more diverse range of tools for drawing and interacting with it. As a result, it is rapidly becoming a popular tool for distributing animations and visualisations on the web, and has seen a large rise in tools and libraries which make manipulating it simple. In particular, we will use JavaScript, and a library for Javascript called jQuery to write to and manipulate our canvas.

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.

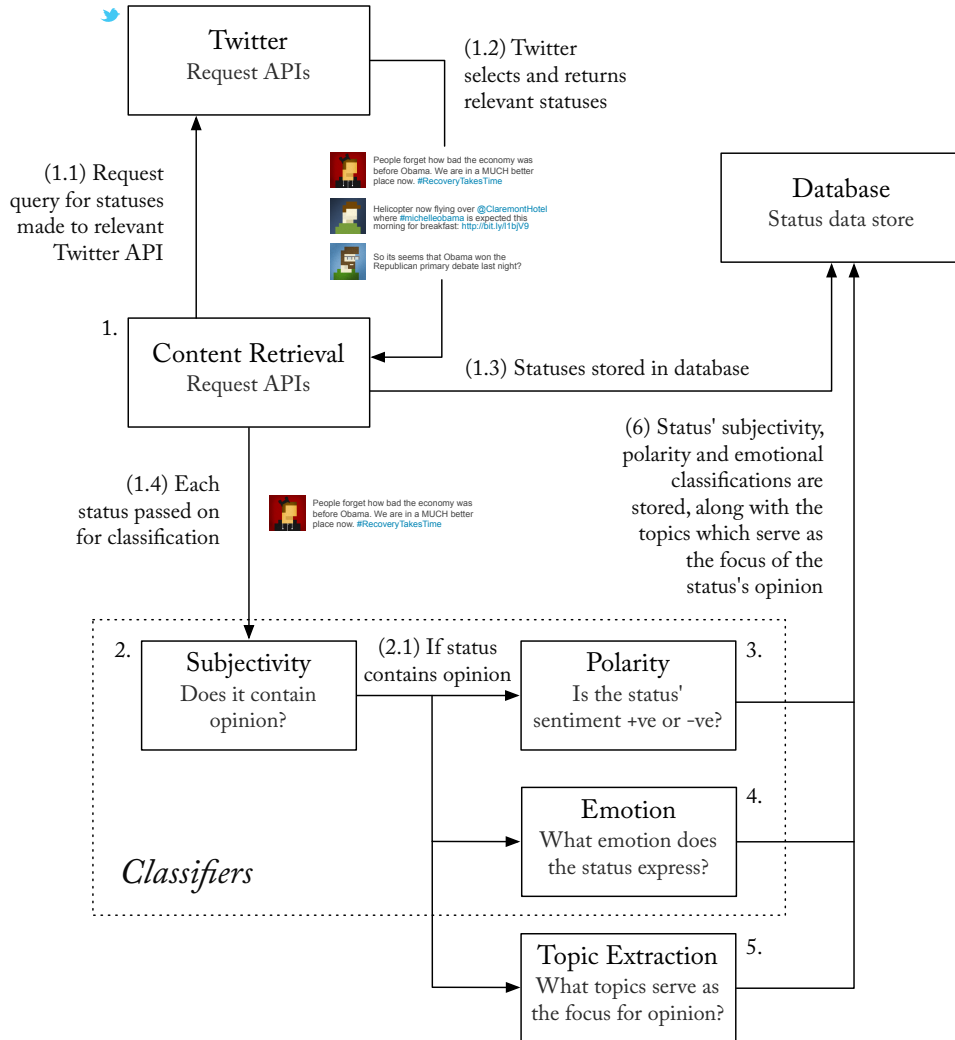
3.1 Design

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 topics 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 the visualisation engine.

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

Figure 3.1: Structure for sentiment analysis engine



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

Within our discussion of the project's implementation we shall sometimes refer to one of the four example tweets below. This is in order to help better illustrate the classification process within our system.

<i>No.</i>	<i>Text</i>
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.	30,000 troops coming home from afghanistan says President Obama. - wait and see

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 the hope of better understanding sentiment on Twitter. We will draw these two slightly disparate elements together under the banner of content retrieval. Although their intended use is quite different, the actual data being collected shares much in common. Once collected the data must then be prepared either for training or classification. This chapter shall first examine the Twitter APIs relevant to this project, before going on to discuss how they are accessed and used. We will then look at the pre-processing which takes place, along with how we label and annotate our data. Lastly we shall evaluate the success of the system within our evaluation section.

4.1 Data structure and storage

As described above, information regarding statuses¹ serves two purposes within this project, firstly when labelled it can serve as training data, and if unlabelled, it can be classified by our sentiment analysis engine in the hope of better understanding the overall opinion and emotion on Twitter. Essentially this means that every status retrieved can be expanded upon with additional information pertaining to both an annotator and-or classifier's decisions as to its sentiment labels. Throughout this project we shall use MongoDB as our primary data store, both for reasons we shall give throughout the rest of this chapter and those outlined in chapter 2. 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 additional attributes.

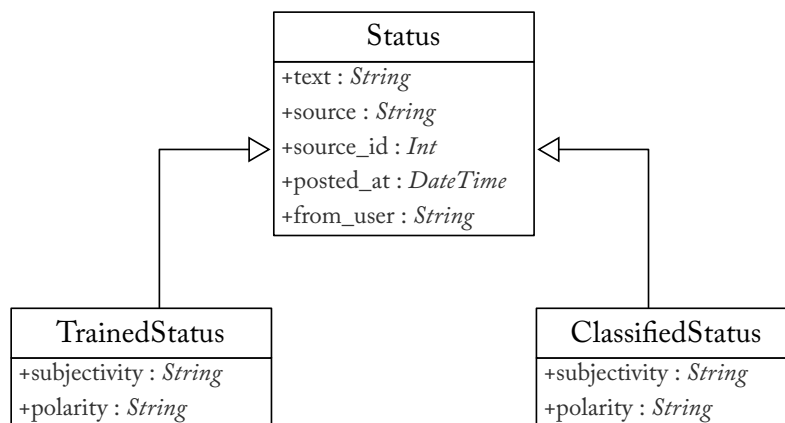
¹We will use the terms *status* and *tweet* interchangeably throughout this project to refer to the same concept. Not only do Twitter use the term status within their object model, but it is also semantically more accurate for this project which eventually hopes to classify statuses from a variety of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

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

```
1 {  
2   "text" : "Hi, this is an example tweet!"  
3   "source" : "search_api" | "streaming_api"  
4   "source_id" : NumberLong("78133750344597504"),  
5   "posted_at" : "Tue Jun 07 2011 17:18:46 GMT+0100 (BST)",  
6   "from_user" : "joeroot"  
7   "trained_status" : {  
8     "subjectivity" : "subjective" | "objective" | "spam"  
9     "polarity" : "positive" | "negative" | "neutral"  
10  }  
11  "classified_status" : {  
12    "subjectivity" : "subjective" | "objective" | "spam"  
13    "polarity" : "positive" | "negative" | "neutral"  
14  }  
15 }
```

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 4.1, 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 MongoDB 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 called *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:

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.

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.

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. Its search method takes a query string and a parameters hash². 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 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 its 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 caching, none provide support for database persistence. Instead our choice of MongoDB and *MongoMapper* mean that simply through retrieving the JSON results and using them 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](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.

In order to collect and store data from Twitter's "*garden-hose*" streaming API we

²It is common practise amongst Ruby developers to use a *parameters hash* for passing in optional arguments to a method. The hash is declared last in a method's arguments, and can be left out when calling the method. If left out, the method definition ensures that the parameters hash is instantiated as an empty hash to prevent any nil errors.

created a singleton *TwitterStream* class. Its `open_stream(track)` method setups a persistent connection to Twitter through the `open-uri` library. All incoming data is cached and whenever a new line symbol is processed, the data is removed from the cache and written to MongoDB. As with our search API, each status' JSON data is amended with a `source` attribute to indicate that the status was retrieved using the streaming API. As our project does not require the streaming API to instantiate Status objects, it simply needs add the JSON to MongoDB.

Similarly to our search API, although there are a plethora of streaming libraries available for Ruby, they are all too bloated for the purposes of this project. Our lightweight implementation has a low footprint and runs smoothly, thus serving the requirements of our project perfectly.

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 tagger³. 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 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 $\frac{1}{3}$.

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

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

³<http://search.cpan.org/~acoburn/Lingua-EN-Tagger>

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 $\frac{1}{3}$.

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

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.

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

Our *TweetTagger* class replicates the above behaviour in Ruby. Colburn's pre-trained probabilities are stored in two separate text files and read into two hashmaps when a *TweetTagger* object is instantiated. In order to tag a body of text with it's appropriate part of speech tags, we can call the `fetch_tags(text)` method, passing in the body of 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 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 exist within our lexicon of trained probabilities, and thus no probability score can be calculated for it, it is passed on to the `unknown_word(word)` method. This attempts to find the word elsewhere, and will be of particular use when adapting the tagger for use on Twitter statuses, and is discussed in detail in section 4.3.2. For now however, we will assume it manages to find the correct tag.

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 our *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, thus if the actual word cannot be found in the word-probability lexicon, we assign it probabilities 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 expression⁴ corresponding to a potential identity through a series of *if-else* statements. Statements higher up the list take priority, thus for identities which provide certainty, such as numbers, we place their condition higher up the *if-else* chain. In order to better illustrate this, example code from the method itself is included in listing 4.4.

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

```

1 def classify_unknown_word(word)
2   if /-?(?:\d+(?:\.\d*)?)|\.\d+\z/ == word
3     classified = "*NUM*" # Floating point number
4   elsif /\A\d+[\d\/:-]+\d\z/ == word
5     classified = "*NUM*" # Other number constructs
6   elsif /\A-?\d+[w+\z/o == word
7     classified = "*ORD*" # Ordinal number
8   elsif
9     ...
10  end
11  return classified
12 end

```

In order to determine whether a word is in fact a URL, an additional condition is added to the `unknown_word(word)` method in order to check whether it 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 its own word and tagged as a URL, as demonstrated 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 words following our new tag within the *tag-probability* lexicon. The new entry into the lexicon is used for calculating the most appropriate tags for any word directly following a URL. URLs are typically used as nouns when not at the end of sentences, thus rather than rebuilding our lexicons with a small corpora of tweets, we chose to simply assign the same next tag probabilities to URLs as we have for nouns.

Listing 4.5: Part of speech tagging for Example 2, taking into account URLs as potential tags.

```

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", "

```

⁴Explain regular expressions

```
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. Mentions represent a unique entity, thus rather than needing a new tag, they are and can be tagged as proper nouns. Essentially they are used as replacements for names; therefore 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 additionally 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 attribute, *mention*, 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}
}
```

Our approach to identifying mentions is similar to that used for URLs. As no word beginning with "@" exist within our lexicon, it is safe to assume all mentions will be passed on to the `unknown_word(word)` method. Within this method, an additional condition is added for identifying words which are mentions. The word is accordingly tagged as a proper noun, and its 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. Instead if processed correctly, they can often be tagged as words with a corresponding natural part of speech. Frequently users will amend keywords within their statuses as hashtags without interrupting the flow of the status. This can be 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 a status' target. Evidently, understanding a hashtag's genuine part of speech is important if we hope to truly determine its sentiment.

The *TweetTagger* class approaches this by first ammeding all hashtag-words' meta object. This is done by setting the meta object's *hashtag* attribute to true and it's *original* attribute to the actual 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 as 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. This enables us to find its genuine part of speech tag by observing its position within the status text.

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. Our labelling system aimed to provide as simple and seamless an interface for annotating data as possible, whilst also remaining open and flexible to changes in labelling needs.

Our labeller offers three core services:

/search enables new tweets for classification to be searched for by the user. This makes use of our earlier *TwitterSearch* API. Figure A.1 contains a screenshot of the form itself.

/train presents all unlabelled tweets for labelling. This only presents Tweets whose source indicates they were found with the *TwitterSearch* API. Our backend

handles correctly formatting the labelled data along with writing it to MongoDB. Our front-end makes use of AJAX techniques. This means that rather than the user having to reload the page every time they classify a tweet, it is handled seamlessly within the browser. Figure A.2 contains a screenshot of an individual un-labelled status.

/trained/:filter presents all labelled tweets for examination or correction. Using the same layout as our /train page, this allows for editing of trained statuses. The optional **:filter** parameter allows for the user to filter the results they see. For example passing in **polarity="positive"** displays only positive trained statuses.

The labellers implementation proved to be simple and efficient for classifying statuses quickly. Furthermore its simple layout meant that adding fields for additional annotation later on within the project proved simple. Our approach to storing labelled data again proved effective, and ensured that the process of adding additional label attributes later was simple.

4.5 Evaluation

As the elements discussed within this section are slightly disparate, we shall evaluate each component individually.

Data structure and storage - We felt that our methods for storing data were flexible, and found that as components and attributes changed throughout our project, the adaptable nature of MongoDB and our approach made change much simpler. Furthermore our classes were simple to use, and made interfacing with Twitter statuses seamless.

Retrieval - Both APIs provided stable and simple methods for retrieving content and storing it.

Pre-processing - Our part of speech tagger proved fundamental in the success of the project. It performed quickly and handled the nuanced complexities presented by Twitter. Conversely, we felt that the underlying statistical data used in predicting tags was not as appropriate for our Twitter domain. Future work might look at training the tagger using part of speech labelled Twitter statuses instead.

Labelling - The interface provided for labelling data was simple and easy to use. It facilitated quick data entry and was easily adaptable as changes in requirement progressed throughout the project.

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 either a supervised or semi-supervised approach to learning for all three classifiers. As a result, our classifiers' implementations share much in common. In particular, this chapter shall focus on how this commonality can be 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. 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 discuss our approach. After this we will summarise the methods discussed and implemented within our class, before finally evaluating overall 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:

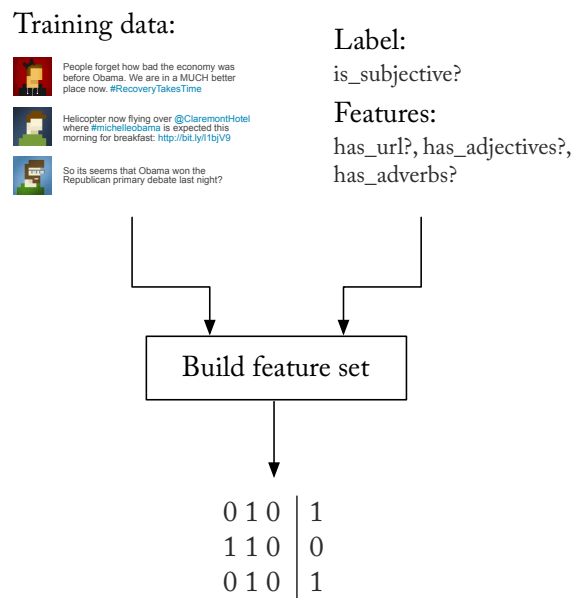
1. *Simple feature selection* when initialising a classifier. For example, we might want to initialise a classifier with just *feature one* before initialising another classifier for a performance comparison, with both *feature one* and *feature two*.
2. *Simple classifier method selection* when initialising 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* with which we are looking to classify statuses, as explained 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 symbol¹ 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 must be implemented as object level methods within any class extending our *Classifier*

¹*Symbols* are Ruby's approach to efficient *String* usage. In our case, rather than initialising a new *String* object every time we want to refer to a certain method, a symbol is only very initialised once but can be used throughout our code. Essentially a symbol can be used as a low footprint replacement for *Strings*.

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.

Listing 5.1: Method for building a status' feature set

```
1 # @features = [:feature_1,...,:feature_n]
2 def build_status_feature_set(status)
3   return @features.map{|f| self.send(f, status)}
4 end
```

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 an array containing each of the training statuses' appropriate label, such as their polarity or subjectivity. It is the values of this label which the 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 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 containing feature sets and one containing labels, we can train our classifier. As explored by Pang et al. (7), the classification method used can impact the effectiveness of our results. We decided to experiment with both the leading probabilistic and non-probabilistic methods, these being the *Naïve Bayes (NB)* classifier and the *Support Vector Machine (SVM)*.

Rather than building our own SVM and NB classifiers, we elected to use two well established libraries, LIBSVM and AI4R, as discussed in section 2.9. As both utilise their own methods and input schemas for training and classification, we implemented an intermediary interface for both the LIBSVM and the AI4R libraries. Both our *NaïveBayes* and *SupportVectorMachine* classes adhere to this interface, 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

```

1 feature_sets = training_statuses.map{|s| self.build_status_feature_set
    s}
2 labels = training_statuses.map{|s| self.fetch_status_label s}
3 svm = SupportVectorMachine.new(feature_sets, labels)
4 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

```

1 def classify(status)
2   feature_set = self.build_status_feature_set status
3   # @classifier is either an initialised SupportVectorMachine or
4   # NaiveBayes object. This is instantiated when initialising our
5   # Classifier object.
6   return @classifier.classify(feature_set)
7 end

```

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

$$accuracy = \frac{|TP \cup TN|}{|TP \cup FP \cup TN \cup FN|} \quad (5.1)$$

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

$$precision = \frac{|TP|}{|TP \cup FP|} \quad (5.2)$$

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

$$recall = \frac{|TP|}{|TP \cup FN|} \quad (5.3)$$

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

$$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall} \quad (5.4)$$

With the definitions of our measures in place, we now had to introduce a suitable method for calculating 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 them, each time 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 test's overall measures. In order to ensure that

training is not biased, we always keep the number of training examples belonging to each label the same.

Although this proved suitable for binary classification, it is not suitable for multi-class classification. Instead for classification with two or more output labels, the *precision*, *recall* and *f-measure* are returned for each potential label. As *accuracy* is already measured across all labels, there was no need for it's results to be calculated for each individual label.

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 methods along with a description of their inputs, outputs and purpose.

Method	Arguments	Returns	Description
<code>initialize</code>	<code>features</code> , <code>label</code> , <code>classifier_method</code> , <code>statuses</code>	<code>none</code>	Initialisation method for creating new Classifier objects. Will initialize and train either a SVM or NB classifier, with the specified features and label, using the labelled statuses as training data.
<code>classify</code>	<code>status</code>	<code>label</code>	The <code>classify</code> 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.
<code>test</code>	<code>k</code> , <code>statuses</code>	<code>accuracy</code> , <code>precision</code> , <code>recall</code> , <code>f-measure</code>	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.

<code>repeat_test</code>	<code>k, statuses,</code> <code>n</code>	<code>accuracy,</code> <code>precision,</code> <code>recall,</code> <code>f-measure</code>	Runs <code>test(k, statuses)</code> <code>n</code> times, returning the average accuracy, precision, recall and f-measure across those <code>n</code> repetitions.
--------------------------	---	---	--

5.6 Evaluation

The *Classifier* class proved fundamental within this project both as a class and a tool. It has allowed for the remainder of our implementation to focus on 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 made 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 (15), although unsupervised approaches' precision rates are high, achieving high recall rates is remarkably difficult. As a result, the problem now lies in designing the elements which will best enable our supervised classifier to separate statuses into their correct labels.

In this chapter we shall first examine how we built our training set, and the decisions 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. Furthermore in assembling our training set, we also want to be able to annotate it so as to better explain our decisions. Although we have already discussed our approach to labelling data in chapter 4, we shall examine what aspects of our approach particularly relate to subjectivity classification within the remainder of this section.

Our approach to subjectivity classification takes a two label approach of either *subjective* or *objective*. Thus the first annotation we want to make to our *Trained-Status* class is one denoting subjectivity. Accordingly each *TrainedStatus* is given an subjective attribute, which takes a value of either "t" or "f". Alongside this we want to collect a more detailed picture of why exactly the status is subjective. In order to do this we collect the phrases which have implied subjectivity in a phrases

array. Although for the purposes of polarity classification we will eventually have to divide our phrases array into two, for now all we are interested in is collecting subjective phrases, but not annotating them with any additional sentiment detail.

With each status now annotated with its subjectivity label and an array of any subjective phrases, we can focus on building our feature set and training our classifier.

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 our own new or adapted ones. In this section we shall examine why and how we implemented our chosen features, but will save a discussion of their effectiveness and final selection until later in this chapter's results and evaluation sections.

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, we now need to extract those words which are used as adjectives. 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 a specific tag, such as `"jjr"` or `"adr"`, and returns its more general tag, in this case `"adj"` or `"adverb"`. This is done by comparing the specific tag against regular expressions corresponding to the general tags. These 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 can focus on how we collect those adjectives for any given status. As this is a method corresponding to an attribute of our status, i.e. its adjectives, we decided to implement this as an `adjectives` method within our *Status* class, as included in listing 6.1.

Listing 6.1: Status object's adjective method for returning all parts of speech used as adjectives

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

It is important to note that the design decision to keep methods such as adjective collection within the *Status* object is consistent throughout our implementation. 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 implemented in a similar to that taken for adjectives. We first implemented a *urls* method for all *Status* objects, implementing it in a similar fashion to the *adjectives* method. Using this, we were then able to 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 Mentions

6.2.4 Subjective clues

As originally observed by Wiebe and Riloff (13), 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 (13). Alongside this we use our own lexicon of subjective phrases, by collecting our phrase annotations as described in section 6.1. The resultant clue lexicons are stored in regular text files, with each line consisting of one clue, as demonstrated 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*, while the `len` field denotes the length of the clue. The `word`, `pos` and `stemmed` fields represent the properties of each word in the clue's phrase, with `stemmed` indicating 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. This is due to "block" being the stem of "blocks". 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. An example item from the resultant hash is demonstrated in listing 6.3.

Listing 6.3: Ruby hashmap representation of listing 6.2

```

1 clues["block"]
2   => {[
3     { :type => "weaksubj", :len => 1, :pos => ["noun"], :stemmed => ["n
4       ], :priorpolarity => "negative"},
5     { :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 and array of words along with their 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 can 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 strong clues.

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

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

6.2.5 Capitalised words

As observed by Barbosa and Fang (21), objective statuses tend to contain significant capitalisation. This is often a result of the status either being spam, or as increasingly the case, it is due to the status containing the HTML page title of the URL it is linking to. Accordingly, we experimented with two features based upon this:

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

$$c.w.f = \frac{|words_{capitalised}|}{|words|} \quad (6.1)$$

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.

$$c.l.f = \frac{|letters_{capitalised}|}{|letters|} \quad (6.2)$$

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

In this section we present our subjectivity classifier's performance across a variety of tests. In doing so we will observe how individual features and combinations of them perform, along with exploring the effect of different classifier types on overall performance. All tests are performed using three-fold cross validation over a set of 300 labelled tweets, and repeated 30 times to ensure an accurate measure.

6.3.1 Individual feature performance

In deciding a suitable feature set for our classifier, we first wanted to observe which features perform best on their own. In order to do this each individual feature was used to initialise our classifier, before running our test methods on each single-feature classifier. In order to truly understand the performance of each feature, we decided to focus on their *accuracy*, *precision* and *recall*. For each measure we wanted

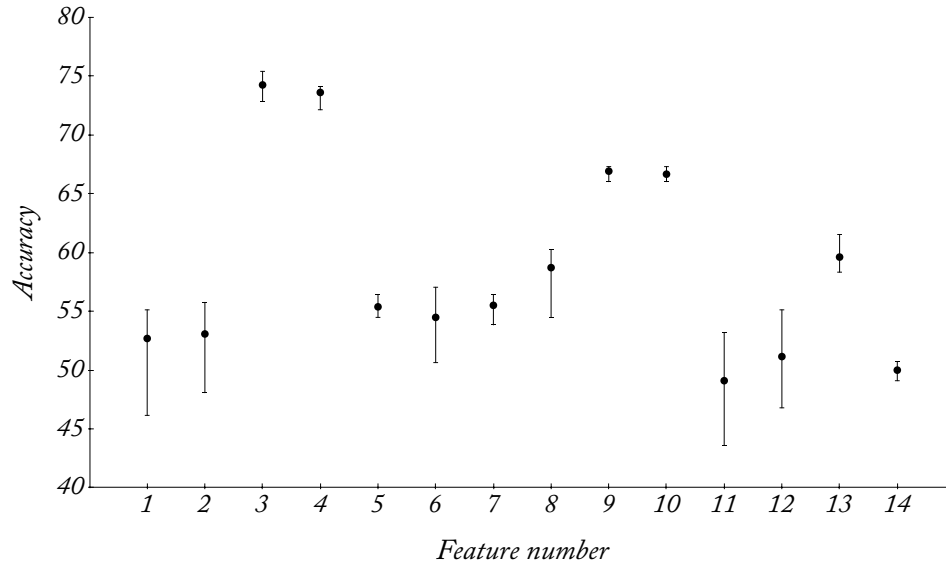
to look at not only the average, but the spread of results across the 100 repetitions. This was so as to provide a better understanding as to the sensitivity of each measure to changes in its training data. We have presented our results for each measure below, and we shall now examine the results for each measure in turn.

It is important to note that in order to present our data in a tidier fashion, we will use the following numbering system when discussing and presenting features:

No.	Feature	No.	Feature	No.	Feature
1	has_adjectives?	7	has_clues?	13	capitalised- _words- _frequency
2	no_adjectives	8	no_clues		
3	has_urls?	9	has_strong_clues?		
4	no_urls	10	no_strong_clues	14	capitalised- _letters- _frequency
5	has_mentions?	11	has_weak_clues?		
6	no_mentions	12	no_weak_clues		

In figures 6.1, 6.2 and 6.3 we present the test results for each of our single-feature classifiers.

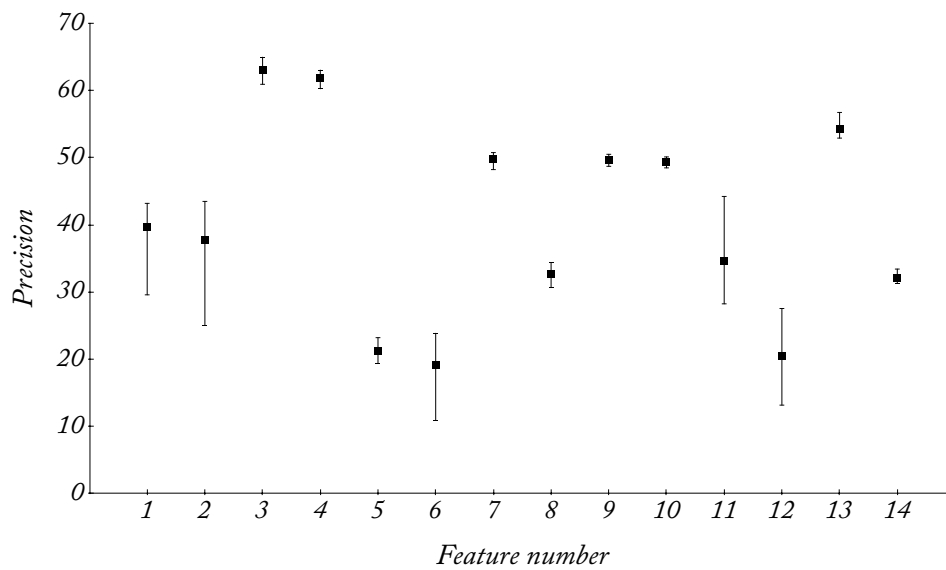
Figure 6.1: Accuracy average and spread for each individual feature



We found that URLs and strong subjective clues were the most informative feature in terms of accuracy. URLs are clearly the most discriminative feature, and when used as a presence-based feature heralded an average classification accuracy of 72.84%. The presence of strong subjective clues also served as a particularly discriminative feature, classifying 66.9% of examples correctly when used as a presence-based feature. Furthermore, when used as individual features, we found

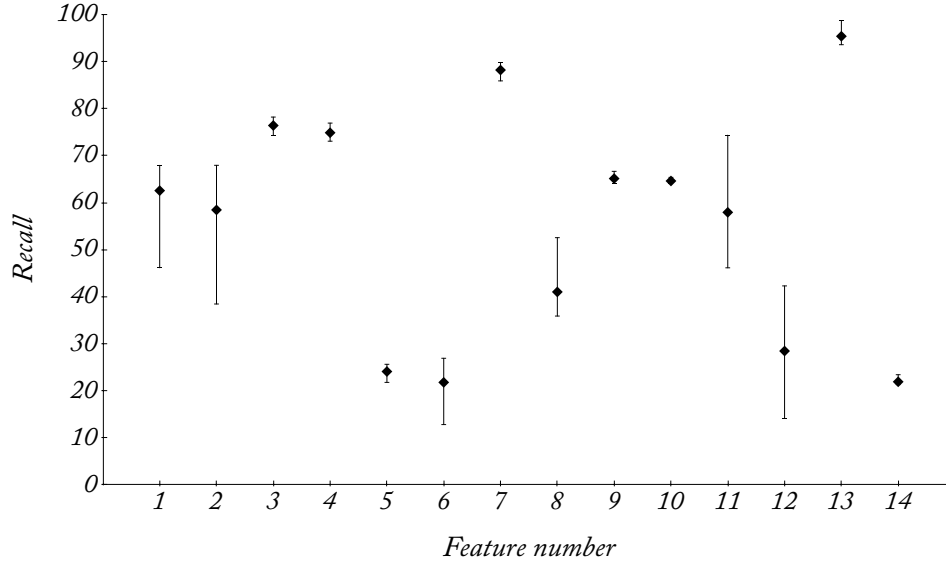
that URLs and strong subjective clues presented little spread in our accuracy results. This suggests that unlike features such as adjectives or weak subjective clues, URLs and strong subjective clues are not sensitive to changes in training data. Interestingly however, we also found that there was little change in accuracy when basing a feature on presence or number of occurrences within the status. All our features, other than weak subjective clue presence, resulted in accuracies greater than 50%. As such, although some do little better than simply guessing, most offer at least some additional insight when classifying a status' subjectivity.

Figure 6.2: Precision average and spread for each individual feature



In general, we found that features on their own presented little in the way of precision. As with accuracy, URLs again offered the best as far as precision rates were concerned, with precision rates of 63.1% when used as a presence-feature and 61.8% when the number of occurrences was taken into account. Also of note was our capitalised words feature which achieved a precision rate of 54.3%. Again weak subjectives performed weakly as a feature, and both it and adjectives suggested that their sensitivity to changes in training data made them unreliable.

Figure 6.3: Recall average and spread for each individual feature



Recall rates proved much better for our features. This is probably largely due to the one sided-ness of much of their classification, especially in the case of capitalised words, which labelled almost 100% of its test data as subjective. Of note however are the high recall rates achieved by URLs and strong subjective clues, which we had already identified as strong potential features. Again adjectives and weak subjective clues demonstrated their tendency to be strongly affected by changes in training data, suggesting their potential unsuitability for subjectivity classification. For adjectives, this is strongly at odds with prior research in the field, and as a result we will further experiment with it when testing how our classifier performs with groups of features.

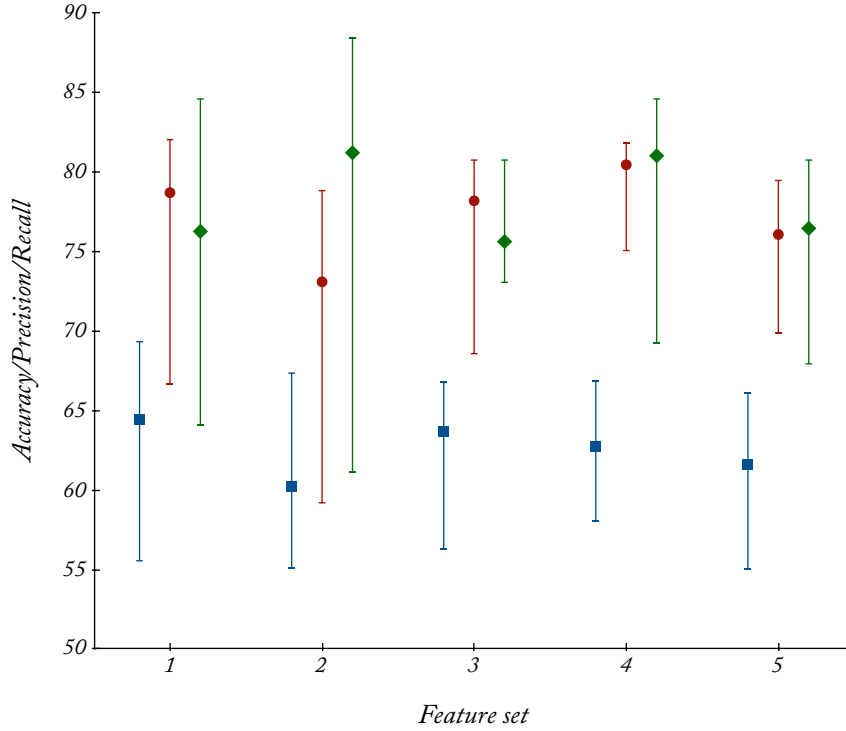
6.3.2 Feature set performance

As a result of our individual feature analysis, we put forward five feature-combinations for further testing:

1. has_urls?, has_strong_clues?
2. has_urls?, has_strong_clues?, no_clues
3. has_urls?, has_strong_clues?, capitalised_words_frequency
4. has_urls?, has_strong_clues?, no_clues, capitalised_words_frequency
5. has_urls?, has_strong_clues?, no_clues, capitalised_words_frequency, has_adjectives?

The results of our tests using the five feature sets are listed in figure 6.4. Blue bars represent precision, red represents accuracy and green represents recall.

Figure 6.4: Precision (square), accuracy (circle) and recall (diamond) results for our five different feature sets.

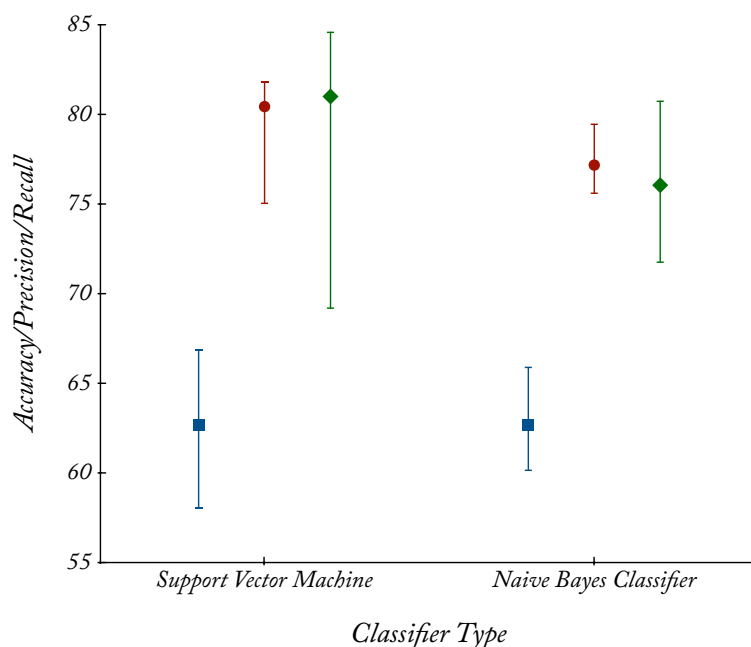


All classifiers perform well, with our fourth feature set performing strongest in terms of accuracy achieving an average of 80.5%. It performed marginally worse than feature set two with regards to precision, scoring an average of 81.0%. Although its precision was not the best, its strong accuracy and recall, combined with its low sensitivity to changes in the training set, make it our preferred feature set. Interestingly adjectives seemed to have an adverse effect on our overall accuracy, precision and recall.

6.3.3 Classifier type

In order to determine the most appropriate classifier type, we ran our tests using a Support Vector Machine for one test and a Naive Bayes Classifier for the other. When building our classifiers, we used the fourth feature set from section 6.3.2. We present our results in figure 6.5.

Figure 6.5: Precision (square), accuracy (circle) and recall (diamond) results for SVM and NB classifier.



In terms overall performance, there is little difference between our SVM and NB performance. Overall SVM slightly outperforms NB in terms of accuracy with an average of 72.7% against the NB average of 71%. We feel however that in the future if the amount of training data we have is to increase, the SVM would provide a more robust and useful classification method.

6.4 Evaluation

Our subjectivity classification rates were strong. With an average accuracy of 80.5%, our classifiers performance further improved upon results typically seen in the field, such as the average accuracy of 70.2% seen by Wiebe et al. (12). The class itself was both simple to use, and train, though this was largely due to the success of our *Classifier* class implementation. We used a good blend of both linguistic features and Twitter meta-features such as URLs, and as a result saw strong classification rates.

In future work there are five areas in particular which we would look to for further improvement,

Training set size - Overall we felt that the limited size of our training set limited our classifiers ability. In future work we would spend more time building up a much more significant amount of labelled data.

Word-sense disambiguation - Although we use part-of-speech tagging to distinguish between the grammatical usage of words, word-sense disambiguation may have seen a further improvement in results. In distinguishing between the senses of words, we provide ourselves with a more detailed account of their use and are thus better informed when classifying subjectivity.

No URL feature - The presence of URLs proved to be a highly discriminative feature when classifying subjectivity. Unlike most other subjectivity classifiers which never come across URLs within their domain, we were able to use this as a feature. We felt that as this was such a dominant feature, its usage both benefitted our classification, but perhaps decreased its linguistic credibility. In future work, we would like to experiment with other features so as not to use URLs which we felt biased our classifier.

Objective, but opinion - In examining our objective statuses, we found that there were occasionally objective statuses and links, which although objective, inferred opinion. In future work we hope to examine ways in which a third classification might be introduced for objective statuses which imply opinion.

Spam classification - Within our current implementation spam is labelled as objective. As the purpose of our classifier was to help identify opinion, this wasn't a problem, however in future work if we want to make better use of our objective data, a third spam label would be an interesting addition.

7

POLARITY CLASSIFICATION

Given that we can now determine whether a status contains an opinion, the next stage within our sentiment analysis engine lies in determining the status' polarity. Polarity classification specifically looks at determining whether a status is *positive*, *negative* or in some cases *neutral*. Approaches to classifying polarity have typically been supervised. This is largely due to the difficulty of identifying the numerous nuanced linguistic details which could imply polarity. Although some (3) have attempted un-supervised approaches, recent work such as that covered by Liu (1) shows that there is a much greater deal of success when supervised techniques are applied. Accordingly, we elected to take a supervised approach for polarity classification, with the hope that we might also be able to draw upon some of the linguistic insight offered by unsupervised approaches.

In this chapter we shall first examine how we labelled and annotated our training set, before going on to explore our choice of potential features and their implementations. Next we shall discuss the results of our testing and explain our choice of features, before finally evaluating the success of our classifier with particular respect to prior research.

7.1 Training set

Our approach to polarity classification will take a two-label approach similar to that proposed by Pang et al. (2). Thus each *TrainedStatus* object will be annotated with a polarity attribute of either "positive" or "negative". Furthermore, we will expand upon our original subjectivity annotations in order to collect phrases which are not only subjective, but positive or negative. In order to do this, we will define two additional attributes `positive_clues` and `negative_clues`. Both of these will be used to store any positive or negative phrases that might occur within the status being annotated.

Thus, for each *TrainedStatus* object, we now have three additional attributes, `polarity`, `positive_clues` and `negative_clues`.

7.2 Features

In order to build an accurate polarity classifier, selecting discriminative features is fundamental. Within our approach we shall draw upon those which have been shown to be successful within literature. Furthermore we shall also look at experimenting with aspects of un-supervised techniques as new and previously untested potential features. The remainder of this section shall focus on our feature implementation, and we shall save a discussion of their effectiveness till our results section.

7.2.1 Unigrams

As noted by Pang and Lee (2), unigrams can often serve as strong discriminative features when classifying polarity. In effect unigrams provide a presence feature for all possible words, however their implementation means that this set of words is limited to those seen when training the classifier, rather than the actual set of all possible words.

As the implementation for unigrams is so distinct from other features, its feature code was built directly into the *Classifier* class, rather than our *PolarityClassifier* class. In order to build our unigrams, we require a two stage process. Before training occurs our unigram set is initialised by iterating through each status, adding its words to our unigram set. When added, words' are down-cased in order to avoid adding the same word multiple times. Once finished we have an array consisting of every single word used in all of our training examples.

When building a status' feature set either for training or classification, we make use of the previously built array of unigram words. For every word in the unigram array, a feature is added to the status' feature set denoting whether the said word occurs within the status. In order to do this we built a `parse.unigram(status)` method, which when given a status, will return the corresponding unigram feature set, as shown in listing 7.1.

Listing 7.1: Classifier class' parse_unigram method.

```
1 def parse_unigrams status
2   # downcases and collects every word within the status
3   words = status.parts_of_speech.map{|p| p["word"].downcase}
4   # iterates over the array of unigram words, noting whether the
      unigram exists within our status' set of words
5   self.unigrams.map{|u| words.include?(u) ? 1 : 0}
6 end
```

Alongside the `parse.unigram(status)` method, we introduced two other methods. The `parse.hashtag.unigram(status)` produces a unigram feature set in which

only hashtags are used as unigrams feature. An additional, more detailed method, `parse_pos_unigram(status)` creates a feature for every word and part of speech tag combination.

In order to simplify the process of including unigrams in our feature set, our classifiers can be initialised with any combination of the three unigram-based features below, just as we would with a normal features. If any of them are included, their corresponding method is called, and the resultant feature set is added to the status' overall feature set. These three methods are:

unigrams is our primary unigram feature. It calls the `parse_unigram(status)` for any given status, and appends the resultant feature set to the status' overall feature set.

hashtag.unigrams builds a unigram feature set which only acknowledges hashtags as words. It calls the `parse_hashtag_unigram(status)` for any given status, and appends the resultant feature set to the status' overall feature set.

unigrams.pos builds a more specific unigram feature set than **unigrams** in which each unigram is a combination of both the word and part of speech it represents. Thus, rather than only checking each word in a status, it will instead check each word and part of speech combination in a status. It calls the `parse_pos_unigram(status)` for any given status, and appends the resultant feature set to the status' overall feature set.

In order to help clarify how the `parse_unigram(status)` method works, an example has been given in listing 7.2.

Listing 7.2: Unigram parsing for Example 1 using a small unigram set

```
1 # let status = Example 1
2 # let self.unigrams = [think, hate, love, good, bad, strong, weak]
3 parse_unigrams status
4 => [1,0,0,1,0,1,0]
```

7.2.2 Polarity clues

Polarity clues are used to identify terms which express polarised opinion. This is a core feature of most polarity classifiers, although the approaches to determining whether a word is polarised vary. We will again draw upon the work of Wiebe and Riloff (15) using chapter 6's *ClueFinder* class. In addition to using the polarised clues presented by Wiebe and Riloff, we also use our own annotated collection of clues and seed words by additionally loading them into the lexicon.

Furthermore when finding clues within a status we also look to further populate our lexicon. All adjective-conjunct-adjective trigrams are extracted, and if one of the adjectives is contained within our clue lexicon, the other is also added. If the

conjunct is "and", then the new clue is tagged with the same polarity as the existing clue, and if the conjunct used is "but" the new clue's polarity is set to the opposite of the existing clue's. Using this un-supervised technique, we are able to further populate our lexicon with no additional effort. This is run on all input statuses, thus it is assumed that the lexicon will improve with time.

Using our *ClueFinder*'s original `clue_data` method we define an additional four methods for our *Status* object, `weak_positive_clues`, `strong_positive_clues`, `weak_negative_clues` and `strong_negative_clues`. In order to handle negation within these methods, whenever a negation word such as "not" is encountered, all words up to the next grammatical punctuation are inverted. For further implementation detail please refer to listing B.3. Each of these are then used to help define twelve new clue-based feature methods. We will define only the first four, as the last eight are simply derivatives which introduce the concepts of *weak* and *strong* clues.

has_positive_clues? returns a boolean value denoting the presence of one or more positive clues.

no_positive_clues returns one of three values based upon the number of positive 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_negative_clues? as with `has_positive_clues?`, but only noting negative clues.

no_negative_clues as with `no_positive_clues`, but only noting negative clues.

7.2.3 Subjective patterns

In Turney's (3) un-supervised approach, he proposes five grammatical structures for identifying subjective phrases. He suggest that the polarity of these phrases are in turn indicative of their sentence's overall polarity. Although we felt that as an un-supervised approach alone it was not suitable for the project, we decided to explore the possibility of using it as a feature within our supervised classification.

Each of the rules from table 2.1 are assembled as items within a `rules` array. Each rule is encoded as an array of three arrays. The first two inner arrays contain all possible part of speech which the first two words of the phrase *must* be. The last array contains all the parts of speech which the third word *must not* be, as implemented in lines 2 to 8 of listing B.4. We use `nil` instead of a third array when the third word can take on any part of speech. With a rule structure now in place, we can split our status into its individual trigrams. Each trigram is then iterated over, each time checking to see if it matches any of the rules expressed in our `rules` array. Any trigram which matches one of the five rules is added to a `patterns` array, which is returned upon method completion.

Once our patterns have been extracted we can then look for clues pertaining to their polarity. This is done by using the *ClueFinder* class to determine the if and

what the polarity of the first two pattern-extracted words are. Using this we define two pattern-based features¹:

has_negative_patterns? returns a boolean value denoting the presence of one or more pattern with a negative polarity.

has_positive_patterns? returns a boolean value denoting the presence of one or more pattern with a positive polarity.

7.3 Results

Within this section we shall take a similar approach to testing as that seen in chapter 6's results section. Again, all tests are performed using three-fold cross validation over a set of 300 labelled tweets, and repeated 30 times to ensure an accurate measure.

7.3.1 Individual feature performance

As with subjectivity, in order to build the most appropriate feature set we first decided to evaluate the performance of each individual feature on its own. In order to truly understand the performance of each feature, we decided to focus on their *accuracy*, *precision* and *recall*. For each measure we wanted to look at not only the average, but the spread of results across the 100 repetitions. We have presented our results for each measure below, and we shall now examine the results for each measure in turn.

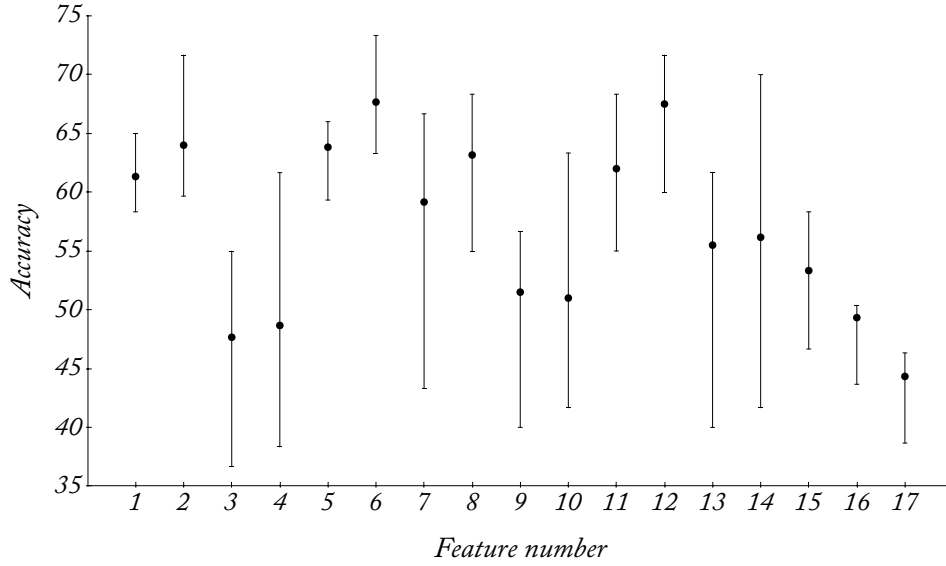
It is important to note that in order to present our data in a tidier fashion, we will use the following numbering system when discussing and presenting features:

No.	Feature	No.	Feature
1	has_positive_clues?	9	no_weak_positive_clues
2	has_negative_clues?	10	no_weak_negative_clues
3	has_weak_positive_clues?	11	no_strong_positive_clues
4	has_weak_negative_clues?	12	no_strong_negative_clues
5	has_strong_positive-_clues?	13	has_positive_patterns?
6	has_strong_negative-_clues?	14	has_negative_patterns?
7	no_positive_clues	15	unigrams
8	no_negative_clues	16	hashtag_unigrams
		17	unigrams_pos

¹As there is typically no more than one pattern in a status, we have opted to use only presence as a feature and not the number of occurrences.

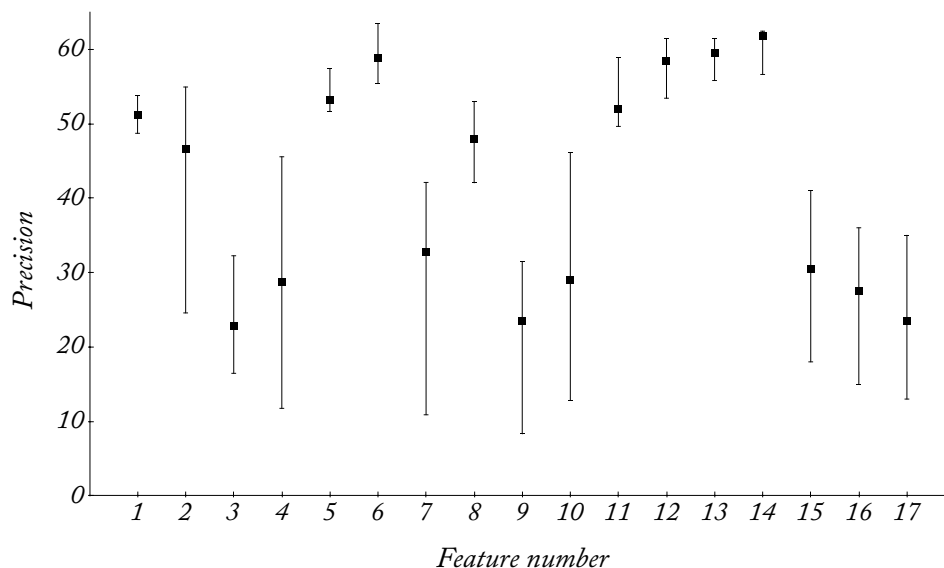
` In figures 7.1, 7.2 and 7.3 we present the test results for each of our single-feature classifiers.'

Figure 7.1: Accuracy average and spread for each individual feature



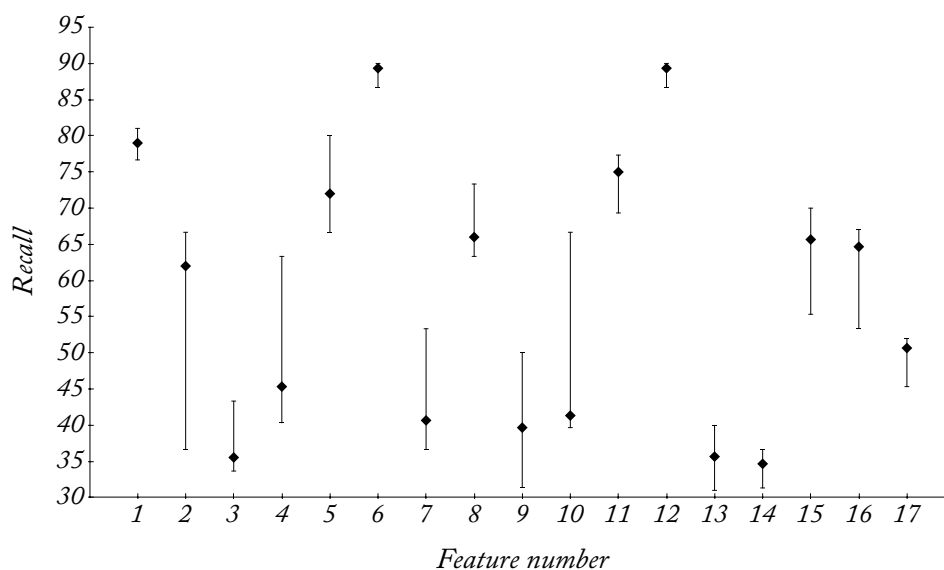
The results across our features is not only varied in terms of accuracy but also in terms of sensitivity to changes in training data. General clue presence along with strong-clue presence/occurrence performed well, achieving strong average classification accuracies of around 65%. Furthermore general-clue and strong-clue presence features both resulted in low spreads, suggesting a resilience to changes in training data. Interestingly, unigrams offer little more than a slight improvement above the baseline expectancy of 50%. Some features such as the two other unigram-based features along with the four variations of weak clues all perform poorly scoring below 50%.

Figure 7.2: Precision average and spread for each individual feature



Our precision results again highlight the benefits of using strong clues as a feature. Furthermore clue presence performs fairly well. Of particular interest however is the high precision rate seen with our pattern matched clues, suggesting that although their accuracy may be low as a whole, they're ability to classify precisely makes them a useful feature.

Figure 7.3: Recall average and spread for each individual feature



Our recall results again highlight clue presence and strong clue-based features as particularly useful within classification. Interestingly, our pattern matched clues have a very low recall, confirming the fact that although their precision is high, their ability to classify slightly more uncertain data is low.

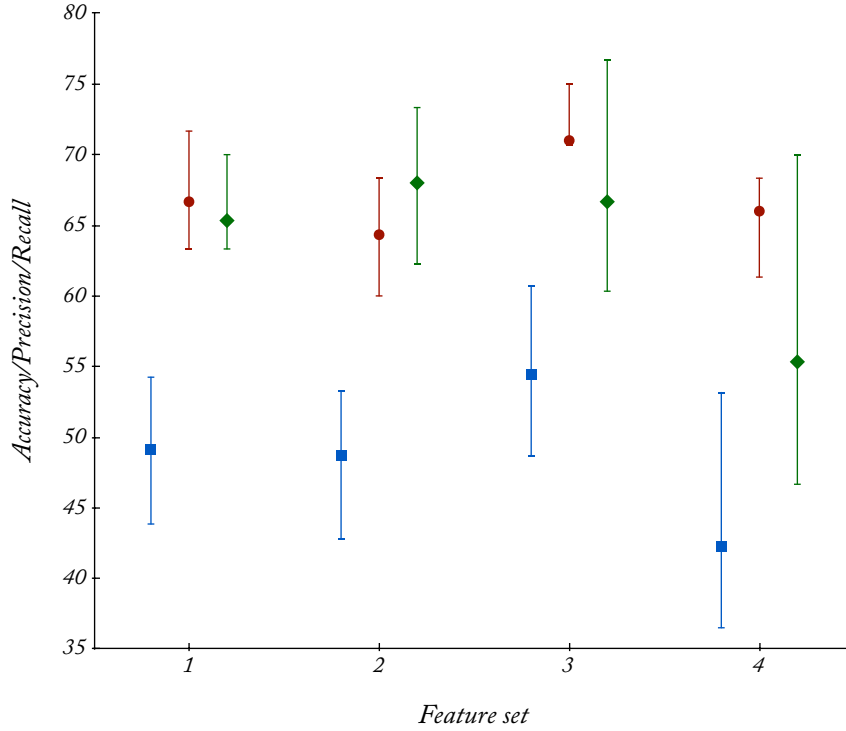
7.3.2 Feature set performance

As a result of our individual feature analysis, we put forward five feature-combinations for further testing:

1. `has_positive_clues?`, `has_negative_clues?`, `has_strong_positive_clues?`, `has_strong_negative_clues?`
2. `has_positive_clues?`, `has_negative_clues?`, `no_strong_positive_clues`, `no_strong_negative_clues`
3. `has_positive_clues?`, `has_negative_clues?`, `has_strong_positive_clues?`, `has_strong_negative_clues?`, `has_positive_patterns?`, `has_negative_patterns?`
4. `has_positive_clues?`, `has_negative_clues?`, `has_strong_positive_clues?`, `has_strong_negative_clues?`, `has_positive_patterns?`, `has_negative_patterns?`, `:unigrams`

The results of our tests using the five feature sets are listed in figure 7.4. Blue bars represent precision, red represents accuracy and green represents recall.

Figure 7.4: Precision (square), accuracy (circle) and recall (diamond) results for our four different feature sets.

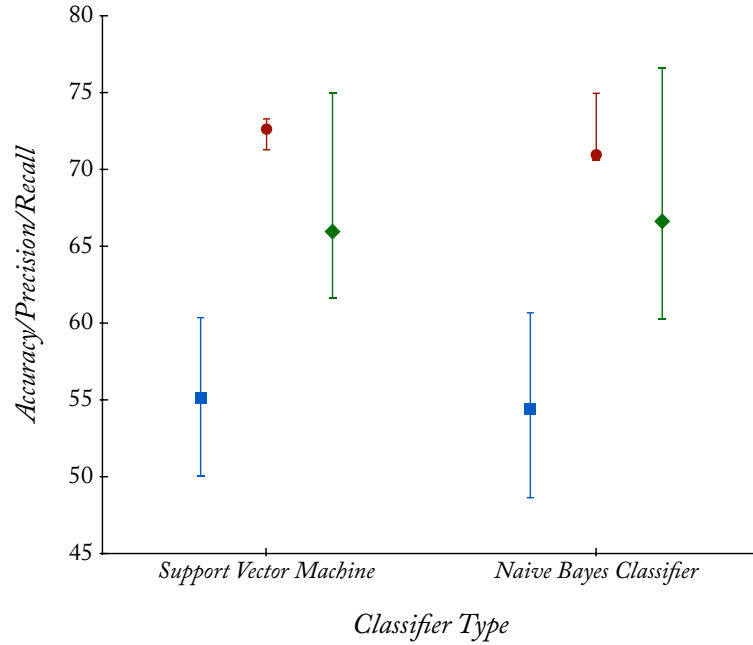


All classifiers perform well, however of the four, we felt that our third feature set performed the strongest. It has the highest average accuracy rate (71.0%), along with a much stronger precision rate. Although its recall is slightly outperformed by our second feature set, overall, we still feel that it's the strongest of the feature sets. Its higher precision can probably be attributed to its use of our pattern-based features which we adapted from Turney's (3) original unsupervised techniques.

7.3.3 Classifier type

In order to determine the most appropriate classifier type, we ran our tests using a Support Vector Machine for one test and a Naive Bayes Classifier for the other. When building our classifiers, we used the third feature set from section 7.3.2. We present our results in figure 7.5.

Figure 7.5: Precision (square), accuracy (circle) and recall (diamond) results for SVM and NB classifier.



Although both the SVM and NB classifiers perform strongly, the SVM's average accuracy of 72.6% against the NB classifier's 71% make it the better classification method. Furthermore our SVM achieves a much stronger average precision than the NB classifier, and only performs marginally worse in terms of recall.

7.4 Evaluation

Overall our polarity classification performed well, achieving an average accuracy of 72.6%. Our results were on par with typical research averages, only being outperformed by Barbosa et al. (21) who achieved rates of 74.9%. We felt however, that this was largely due to differences in training data size. As with our subjectivity classification, we believe that with a larger training set we might have seen an improvement in overall classification rates.

In future work there are a few areas we would look to for further improvement,

Training set size - as discussed briefly above and in chapter 6, we felt that the size of our training set has a negative overall impact on classification rates, and in future would spend more time on assembling a larger set.

Word-sense disambiguation - as discussed in chapter 6, we feel that word-sense disambiguation would in future help further improve our results.

Neutral classification - Our approach to sentiment analysis focussed on positive and negative classification, however on occasions we felt that opinionated data presented no actual polarised opinion. Accordingly, future work might look at how we can introduce a third label for neutrality.

8

EMOTION CLASSIFICATION

With status polarity and subjectivity now classified, we can go on to look at what further understanding of status sentiment can be gained through classifying emotion. Rather than taking a supervised approach as often seen in literature (27), we will approach our emotion classification through a largely un-supervised approach. As our goal is to gain additional understanding as to a status' sentiment, we felt that a precise approach, with easily justifiable results was more appropriate. Techniques such as that put forward by Yang et al. (26), although seemingly accurate, source labelled data for training and testing with no human input. Instead they use emoticons to generate labels which we felt provided significant room for misguided emotion classification. In particular there is no attempt within their work to determine whether the emoticon truly represents the emotion they are ascribing to it. Despite this, we will draw upon elements discussed in supervised approaches, especially with regards to the use of WordNet such as in the approach put forward by Alm et al. (27).

Our approach aims to blend aspects of common lexicon-based classification, with the deeper semantic insight gained through the use of tools such as WordNet. Essentially, we hope to determine whether a word and the grammatical way in which it is being used, can help relate to us any emotion. As WordNet semantically defines concepts of emotion and feeling, it will prove fundamental in our approach. Within this chapter we shall first briefly examine what exactly WordNet is, before going on to discuss our implementation of it. We shall then go on to explore how exactly we used WordNet to generate our lexicon, before examining how exactly it is used in order to determine any emotion a status might contain. Lastly we shall examine the extent to which our approach was a success along with its potential for any further improvements.

8.1 WordNet

WordNet is a large lexical database of and for the English language. Words are semantically grouped amongst each other through *synsets*, each of which denotes a concept of commonality. The results is a huge, detailed network of interconnected

and semantically intertwined words. In order to grant a better picture as to how this is done, we have defined it's core concepts below:

Words serve as the core entities within WordNet. A *word* entity can however also be a phrase, thus the phrase "*academic year*" for example is considered a word, even though it would not be in the traditional sense of the word.

Synsets serve as sense based groupings for similar words. For example the *synset* grouping together words which describing "*a beloved person; used as terms of endearment*" includes the words "*beloved*", "*dearest*" and "*honey*".

Senses a word can take on any number of senses. Essentially, each *sense* maps any given word to a synset.

Lexicon links link one word-synset pair (or sense), to another word-synset pair. The link itself is semantic, thus if for example it is a *hyponym* link, would imply that a word which is a hyponym of another is in fact a more specific term for the other word. For example "*scarlet*" might be considered a hyponym of "*red*". Other link types denote *similarity* or *hypernyms* (the opposite of hyponyms).

Categories help better define synsets, for example denoting that words being used as a memeber of certain synset might be emotion verbs or location nouns.

WordNet's structure is of particular use within this project. For example, given the word "*adore*", WordNet can proceed to move up and down it's chain of general-ity. In trying to find a more general meaning for the word, we can use WordNet to find its hypernym "*love*". Essentially WordNet provides layers of details, with each increase in layer leading to further generalisation. Thus, if we were to label words at a higher layer with their appropriate emotion classification, given any word, we should be able to move up the layers in order to determine whether it is related to an emotion.

In order to make use of this functionality within Ruby we had to build our own API for accessing WordNet. The WordNet database is distributed in various forms, and for the purpose of this project we chose to use its MySQL distribution. A basic class structure was implemented, in particular focussing on several core methods, noted in table 8.1. All classes inherit from a *Model* class which handles reading from the database. In particular, it offers a `find(params)` method which when given a series of attributes and their corresponding values, will search the database for any matching instances.

Table 8.1: Core WordNet classes and methods

Class	Method	Description
Word	senses	Returns all potential senses in which the word can be used. These are returned as an array of <i>Sense</i> objects

	synsets	Will return each senses corresponding synset. These are returned as an array of <i>Synset</i> objects.
Sense	definition	Returns the senses corresponding textual description. This is returned as a <i>String</i> object.
	pos	Returns the senses corresponding general part of speech, e.g. "adj" or "noun".
	sysnet	Returns the senses corresponding synset. This is returned as a <i>Synset</i> object.
	word	Returns the senses corresponding word. This is returned as a <i>Word</i> object.
Synset	category	Returns the synsets corresponding category e.g. "verb.emotion".
	similar	Returns all sysnets which are semantically linked to as similar. These are returned as an array of <i>Synset</i> objects.
	hyponyms	Returns all sysnets which this synset is semantically linked to as a hyponym. These are returned as an array of <i>Synset</i> objects.
	hypernyms	Returns all sysnets which this synset is semantically linked to as a hypernym. These are returned as an array of <i>Synset</i> objects.
	antonyms	Returns the sysnet's antonym, i.e. the synset whose sense is opposite, e.g. love to hate. These are returned as an array of <i>Synset</i> objects.

8.2 Building an emotion lexicon

In order to classify words according to their correct emotion, a lexicon mapping a certain layer of words to each of our eight emotions is required. In order to do this, we took each of our eight emotions along with their similar words, and defined an emotion lexicon, as listed in table 8.2.

Table 8.2: Core emotion labels and corresponding Wordnet senses

Joy	joy(v), rejoice(v), gladden(v), joy(n), delight(n), pleasure(n), joyousness(n), joyfulness(n), happy(adj), well-chosen(adj), glad(adj), felicitous(adj)
Trust	trust(v), believe(v), hope(v), entrust(v), intrust(v), confide(v), commit(v), trust(n), confidence(n), faith(n), reliance(n), trustingness(n), trustfulness(n), trustful(adj), trusting(adj)
Fear	fear(v), dread(v), worry(v), reverence(v), fear(n), fright(n), frightened(adj), scared(adj)

Surprise	surprise(v), affect(v), impress(v), attack(v), assail(v), act(v), surprise(n), astonishment(n), amazement(n), surprised(adj)
Sadness	sadness(n), gloominess(n), lugubriousness(n), sadness(n), unhappiness(n), sorrow(n), sad(adj), deplorable(adj), distressing(adj), lamentable(adj), pitiful(adj), sad(adj), sorry(adj)
Disgust	disgust(v), revolt(v), nauseate(v), sicken(v), churn up(v), gross out(v), revolt(v), repel(v), disgust(n), disgusting(adj), disgustingful(adj), distasteful(adj), foul(adj), loathly(adj), loathsome(adj), repellent(adj), repellant(adj), repelling(adj), revolting(adj), skanky(adj), wicked(adj), yucky(adj)
Anger	wrath(n), anger(n), ire(n), ira(n), angry(adj), furious(adj), raging(adj), tempestuous(adj), wild(adj)
Anticipation	anticipation(n), expectation(n), prediction(n), prevision(n), expectancy(n), predict(v), foretell(v), prognosticate(v), call(v), forebode(v), anticipate(v), promise(v), expect(v), know(v)

In order to determine if a word and its corresponding part of speech exist within our lexicon, we build a singleton *EmotionFinder* class. Our lexicon is stored within a text file, and is read into a hashmap. Each key within our hashmap represents an emotion, whilst its corresponding values consist of word and part of speech pairs, as demonstrated in listing 8.1.

Listing 8.1: Example key usage for emotion lexicon hashmap

```
# let @@emotions be our lexicon hashmap
@@emotions[:anger]
=>
[
  ["wrath", "noun"],
  ["anger", "noun"],
  ["ire", "noun"],
  ["ira", "noun"],
  ["angry", "adj"],
  ["furious", "adj"],
  ["raging", "adj"],
  ["tempestuous", "adj"],
  ["wild", "adj"]
]
```

Within our *EmotionFinder* class we have implemented a `fetch_emotion(word, pos)` method, which when given a word and part of speech, will check to see if they have a corresponding emotion. If they do the corresponding emotion symbol will be returned, and if not nil is returned, see listing 8.2.

Listing 8.2: EmotionFinder method for determining whether a word and its corresponding part of speech indicate an emotion

```
1 def fetch_emotion(word,pos)
2   return @@emotions.select{|k,v| v.include? [word,pos]}.keys
3 end
```

8.3 Classifying emotion

But with a lexicon defined, how do we set about classifying emotion. As Plutchik's model suggests, emotion can be combined, thus we are not approaching this problem with the hope of producing one single emotion label. Instead, we are hoping to identify all of the emotions expressed. In order to do this we look at each individual word within our status, in the hope of identifying a relationship to an emotion from within our lexicon.

For example, within *Example 4* one might say that the phrase "*wait and see*" contains an element of *anticipation*. But how does our classifier determine this? For each word in our status we first retrieve all senses of the word and its part of speech using our *Sense* class' find method, passing in `{:word => word, :pos => pos}` as parameters. With our senses retrieved, each sense and its corresponding synset is examined. For each word in the synset being examined, we check to see whether it has corresponding emotion by calling our *EmotionFinder*'s `fetch_emotion(word,pos)` method. If no emotion is found, we now examine our current synset's corresponding hyponym synset. We again repeat the process, checking whether any of the synsets words exist within our emotion lexicon. If no emotion is found, this is repeated until we reach a synset with no hyponym, in which case we have reached the top-layer and can generalise no further. Typically our search will have to examine no more than two layers before reaching this top-most layer. It is important to note that negation is handled in a manner similar as to when classifying polarity in chapter 7. Whenever a negation word is encountered, all words up to the next grammatical punctuation are negated. This means when retrieving our first synset, rather than simply returning it, it's antonym is returned instead.

Returning to *Example 4*, the word "*wait*" is used as a verb. Thus when fetching the relevant senses, we would be returned four potential senses. The first two senses' corresponding synsets contain no emotion words, however the third synset consists of the words "*expect*", "*look*" and "*await*", each of which are being used as verbs. When passed on to our *EmotionFinder* class, both "*look*" and "*await*" return nothing, however "*expect*" returns the emotion *anticipation*. Thus our status will have the emotion *anticipation* added to its classification.

8.4 Evaluation

Overall our emotion classifier performed well, however unlike our subjectivity and polarity classifiers, its effectiveness proved far more challenging to evaluate. There are a series of blurred lines between the emotion labels, and as Alm et al. (27) observe, even human disagreement is frequent. Furthermore, as a result of our limited training data and low classification recall rate, automated testing was difficult. In manually checking each emotion, we felt that the classifier performed well. In general we agreed with the majority of classification labels even though they may not have been our first choice. Importantly, as our emotion classifier takes an unsupervised approach, its ability to explain any classification was found to be useful and insightful.

The avenue explored within our research certainly has potential to be expanded upon, and we feel that in particular if two improvements are made, significant progress could be made. The first improvement is, as discussed in previous chapters, word-sense disambiguation. We feel that WSD is most important in emotion classification where differences are often subtle. Along with this, we feel that in building a more accurate lexicon, in which synsets (i.e. groups of words with the same sense) are labelled with emotions, our classifiers performance could be greatly improved.

Overall, we felt that our emotion classifier performed well, and with small tweaks and a concerted lexicon building effort, we feel that significant improvements could continue to be made.



TOPIC EXTRACTION

With our statuses' sentiment now classified, we can now look to better understanding the targets and key themes within it. Our approach to topic extraction hopes to identify key words and topics, thus the term topic extraction is used loosely. Our approach hopes to observe and extract the grammatical patterns of keywords within our training data, and use those in turn to extract key words and topics from previously unseen statuses. Although we do not want incorrect keywords, as our ultimate aim is to identify similar statuses through matching topics, to a certain extent we are less worried in identifying too many topics, than we are with identifying too few.

9.1 Extracting patterns

In order to build up a set of topic extraction patterns, we first had to annotate our data in a suitable manner. We added an additional topics attribute to our trained status objects, allowing us to specify an array of topics. Each topic can be one or more words, but must be annotated as it is represented in the data. For example, if we were to annotate *Example 3*, we might select "Obama", "European debt crisis" and "US" as its core topics and keywords. However in labelling the status, we could not for example use "Europe's debt crisis" as a topic, as it does not represent how the topic is actually expressed within the status.

Our patterns hope to identify the part of speech structures used to express topics within statuses. In order to do this we look to extract the parts of speech used to express the topic, along with the parts of speech for the two words which precede and follow the topic itself. Thus using the topics we selected for *Example 3*, we might observe the following patterns:

1. *Obama*: [nil, "nnp", "vbz"]
2. *European debt crisis*: ["vbz", "jj", "nn", "nn", "md"]
3. *US*: ["nn", "nnp", "rb"]

In order to achieve this, we must first identify each annotated topic within our parts of speech array. This is done by first noting the word length, n of the topic

being identified, before then splitting the parts of speech into its n-grams. With a series of n-length phrases now available, it is simply a matter of iterating through each n-gram and checking whether it matches our topic phrase. If so the parts of speech are noted, and the pattern returned. We wrap this up within the *TopicExtraction*'s `extract_pattern(topic, pos)` method, as shown in listing B.5.

Thus, for every labelled status, we extract its annotated topic's part of speech pattern, building an overall set of rules which help identify topics within previously unseen statuses. We store these patterns in our *TopicExtraction*'s class level array, `patterns`.

9.2 Identifying patterns

With our patterns now identified, we need to define a method to identify them within any given status. This is done by making use the Ruby *Array* object method, `include?(object)`, which returns a boolean value indicating whether an object occurs within an array. Using the maximum pattern length, we create a collection of n-grams for our status using values of n ranging from three to the said length. For each n-gram we then proceed to check whether it's part of speech tags match any pattern within our `patterns` array, using the aforementioned `include?(object)` method. If the n-gram does exist, its first and last elements are removed, and the remaining topic is returned. This method is wrapped up in the class level `extract_topics(status)` method, which returns an array of topics. We have included the method code in listing B.6.

9.3 Evaluation

Our topic extraction module proved an effective way of gathering keywords and topics for statuses. Although difficult to assess quantitatively, we found that in manually examining the results our extraction engine almost always identified the core topics. In some cases it was slightly overzealous in identifying topics, and future work might look at how we can ensure it identifies a slightly more concise set of topics. Nonetheless the results were accurate, and in general did not omit any important topics when attempting to extract them.

As research into topic extraction tends to be carried out for a whole gamut of purposes, it is difficult to assess our performance in comparison to others. Overall however, we felt that our topic extraction module performed well and perfectly suited the needs of our project.

10

DELIVERY

With each of our core components now in place, this final chapter hopes to draw all the individual pieces together, along with discussing how they have been made available.

10.1 Sentiment analysis engine

Our sentiment analysis engine is brought together through the singleton *Engine* class. This class initialises each of our three classifiers along with our topic extraction engine. Once initialised, status objects can be classified by calling its `classify(status)` method, which will return a hashmap containing each of the classifiers' results along with the status' topics, as demonstrated in listing 10.1. The method can handle both *Status* objects and *Strings*.

Listing 10.1: Returned part of speech tags for Example 1

```
1 # let status = Example 1
2 Engine.classify status
3 =>
4 {
5   :subjective => true,
6   :polarity => :positive,
7   :emotion => [:trust],
8   :topics => ["David Cameron", "job", "leader", "coalition", "
               labour"]
9 }
```

Along with returning the results, the hashmap is written as JSON to the status' `classified_status` attribute. Thus the above output is persisted in MongoDB and is referred to from that point forward whenever the status' classification is requested.

When the engine is running, it sets the *TwitterStreaming* API to run in the background, constantly feeding data into MongoDB. Our sentiment engine polls this is

every 10 seconds checking for new unclassified statuses, and if found it will syphon them off to the `classify(status)` method.

10.2 Online API

In order for us and others to make use of our collected and classified data, we offer two core web services. Our first, a classification service, is based upon our *Engine* class. The second, a retrieval service, is built upon our MongoDB store. The web services are made available using the *Sinatra* web framework. Below, is a more detailed outline of each service:

`/classify?text=:string` - when passed a status' textual content, this method will utilise the *Engine* class in order to classify it. The text is formatted as a String object, and passed to the `classify(status)` method. The returned hashmap is formatted as JSON and delivered back to the user.

`/results/:params` - allows the user to query our MongoDB store for classified statuses. With no parameters, the method will simply return the last 200 classified statuses. With parameters, the user can specify from and until dates to retrieve statuses from within a certain time frame. Additionally, the subjective, polarity, emotion and topic parameters each take a comma separated list of acceptable values. For example,

`polarity=positive&topic=David%20Cameron,NHS`

will return all positive statuses regarding David Cameron and the NHS. The results are formatted as a JSON array and returned to the user.

10.3 Visualisation

Our HTML5 visualisation utilises the `results` service above to render an interactive demonstration of sentiment on Twitter. Making use of the *canvas* element, topics are rendered as spheres whose size grows as the number of statuses discussing them increases. Spheres whose topics tend to be discussed together are grouped closed to one another. The colour of the spheres is determined by their polarity and emotion. Using Plutchik's emotion wheel, topic spheres are rendered as in figure 10.1.

Figure 10.1: Example demonstrating our visualisation's spheres and approach to emotion colouring. The labels inside each sphere would typically be a topic, however for the purpose of explanation, we have used the sphere's corresponding emotion label instead.



10.4 Evaluation

We felt that both our web service implementation and visualisation tool met the project's core aims.

Our classification service was simple to use for developers, and managed to mask the complexities of the underlying task well. Similarly, our results service proved effective and its ability to easily filter results made it a viable data source for developers. The results for both services are returned in a simple and open format, thus further helping developers to make use of our engine and results. In developing our visualisation tool, we found that both services' fast response time made them viable tools for use within real-time data processing applications.

Our visualisation tool provided a simple intuitive interface for understanding sentiment on Twitter. In using HTML5 as a platform for delivery we ensured that our visualisation could be easily accessed by all across a wide variety of platforms.

PART III | EVALUATION

11

EVALUATION

As each component has been evaluated in its respective chapter, within this chapter we shall instead evaluate to what extent our original project aims have been met. We shall first examine those discussed within our core aims, before finally evaluating to what extent our secondary aims were met.

11.1 Core aims

11.1.1 Sentiment analysis engine

1. *A Twitter scraper which will fetch live statuses from Twitter.*
Our content retrieval module draws live data from the size in an effective and stable manner. Statuses are pre-processed and stored in MongoDB, whilst our Status class provides a simple interface for accessing the data from within Ruby.
2. *A method for determining whether fetched statuses are opinionated or not.*
Our subjectivity classifier performs strongly in identifying opinionated text, and serves as an effective tool for doing this. We feel that our blend of existing research with Twitter focussed aspects worked well, and our accuracy rates are better than those seen in literature.
3. *A method for identifying whether fetched statuses' opinion is positive or negative.*
Our polarity classifier serves as an effective tool for determining polarity. Its classification accuracy is on par with others in literature, although there are some which marginally outperform ours. We feel that this is largely due to our limited training set size, and future work would place a much larger focus on building this.
4. *A method for identifying the core topics which our fetched statuses are discussing.*
Our topic extraction module performs well within the context of our project, and serves as a useful tool for drawing together statuses whose focus is similar.
5. *A means of persisting our statuses and the results of the above methods, such that they can be easily retrieved at any time.*

As discussed in our first point, the MongoDB store serves as a robust store for our data and the Status class makes accessing its data simple.

11.1.1.2 Web services

1. *Requests can be made to classify a status, upon which the service should return the classification engine's results.*

Our classification web service provided a simple to use way for external developer to make use of our sentiment analysis engine. Response times were fast, and the formatted JSON results made computer interpretation simple.

2. *A method for requesting any classified data our engine may have stored.*

Our results web service provided an easy way of accessing our live, and past, results. The parameters allowed developers to tailor the results they were retrieving, so that only information relevant to them was returned. Furthermore, as with our classification web service, response times were fast and the JSON results made computer interpretation simple.

11.2 Additional aims

1. *Extend our sentiment analysis engine to give a more detailed account of a broader spectrum of emotion.*

Little research has been conducted into emotion classification. We felt that our emotion classifier presented an innovative and unique take on emotion classification. The assigned emotion labels were in general close to our own manual annotations, and we felt that it provided a viable and effective method for classification.

2. *Web based visualisation tool for better understanding and exploring sentiment on Twitter.* Our visualisation tool was an innovative and unique way of exploring opinion. It presented our detailed results in a simple and fluid manner, along with making the results easily accessible to all.

11.3 Summary

In summary we feel that the project met all of its goals with in general a great deal of success. Our subjectivity classifier performed well, and although our polarity classifier was not perfect, we feel that the we have identified the changes required to improve it. Our exploration into emotion classification proved insightful, and the results were promising. Our topic extraction engine serves its purpose within the project and managed to identify core topics well. Additionally, we felt that the engine has a whole was fairly extensible, and our approach to its design means that using it with other micro-blogging services should be simple. Finally, we felt

that our overall delivery methods were simple to use, and managed to mask the complexity of the tasks they were interfacing to well.

12

CONCLUSION

12.1 Contributions

This project's primary contribution is its sentiment analysis engine, which provides a fully working system for determining each component of a Twitter statuses emotion in significant detail. Its subjectivity and polarity classifiers successfully both implement and adapt existing research, along with introducing new ideas and approaches. Our emotion classifier presents one of the first significant attempts at understanding detailed emotion on Twitter, and the research conducted into it proved promising, suggesting that our approach is a valid one for exploring in further work.

Our project's ability to source and process live data make it a valuable tool, and the web services implemented make our data and engine easily available to other developers. Finally the visualisation offers an insightful way of exploring emotion on Twitter, and really helps demonstrate the practical use of understanding sentiment.

12.2 Further work

Word-sense disambiguation - As discussed in chapter 2, word-sense disambiguation looks at how we can understand the sense in which a word is being used. In effect, it looks at taking part of speech tagging one step further by introducing an additional layer of detail. We feel that if done well, a words-sense disambiguation tool could have a dramatic impact on the precision of our classifiers. Our present ability to confuse the sense in which words are being used might be perceived by many to be sloppy, and we too feel that it is an issue. Research into it is limited however, thus further work exploring it would need to be significant.

Training data - Supervised Machine Learning is fundamentally dependant upon a strong body of labelled data. We feel that our project was let down somewhat by our limited data set. When training our classifiers, we found that no overfitting was seen, serving as a quantitate indication that our training set was

indeed to small. In future work, we would spend a much more significant portion of time on building an appropriately sized training set.

Subjectivity classification - Further improvements to our subjectivity classifier might look at how we can build our classifier to be less dependant upon URL based features. We felt that this was a significant bias in our classification and would hope to introduce other features which might alleviate this problem in future work. Additionally we would like to introduce an additional spam label, to help better distinguish our objective data.

Polarity classification - Along with the word-sense disambiguation described above, we would like to approach introducing a third neutral label to our classifier. Furthermore, we would like to explore polarity strength in further work, an area which this project didn't have time to touch upon, but could be of significant use.

Emotion classification - We felt that our classifier opened up an interesting avenue for future work. Alongside word-sense disambiguation, we would like to explore the possibility of labelling WordNet's synsets with their appropriate emotion classifications. We feel this could be a genuine significant step forward, as the level of detail afforded by senses allows for much more certainty when classifying emotion. Ultimately, we believe that this is the most significant improvement which could be made to our work.

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. (2011, May) On chomsky and the two cultures of statistical learning. [Online]. Available: <http://norvig.com/chomsky.html>
- [5] B. Pang, ``A sentimental education: Sentiment analysis using subjectivity summarization based on minimum cuts," 2004.
- [6] I. Rish, ``An empirical study of the naive Bayes classifier," *IJCAI 2001 Workshop on Empirical Methods in Artificial* ..., 2001.
- [7] B. Pang, ``Opinion mining and sentiment analysis," *Foundations and Trends in Information Retrieval*, 2008.
- [8] C.-W. Hsu and C.-J. Lin, ``A comparison of methods for multiclass support vector machines," *IEEE Transactions on Neural Networks*, vol. 13, no. 2, pp. 415--425, Mar. 2002.
- [9] C. Lin and C. Chang, ``A practical guide to support vector classification," *National Taiwan University*, 2004.
- [10] R. Mihalcea and C. Banea, ``Learning multilingual subjective language via cross-lingual projections," ... *MEETING-ASSOCIATION FOR* ..., 2007.
- [11] 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: Association for Computational Linguistics, 1999, pp. 246--253.
- [12] J. Wiebe, ``Effects of adjective orientation and gradability on sentence subjectivity," *Proceedings of the 18th conference* ..., 2000.
- [13] -----, ``Learning subjective adjectives from corpora," *Proceedings of the National Conference on Artificial* ..., 2000.
- [14] -----, ``Subjectivity word sense disambiguation," *Proceedings of the 2009* ..., 2009.

- [15] E. Riloff and J. Wiebe, "Proceedings of the 2003 conference on Empirical methods in natural language processing -," in *the 2003 conference*. Morristown, NJ, USA: Association for Computational Linguistics, 2003, pp. 105--112.
- [16] J. Wiebe, "Word sense and subjectivity," ... *of the 21st International Conference on ...*, 2006.
- [17] -----, "Recognizing contextual polarity in phrase-level sentiment analysis," *Proceedings of the conference on ...*, 2005.
- [18] F. Benamara and C. Cesarano, "Sentiment analysis: Adjectives and adverbs are better than adjectives alone," *Proceedings of the ...*, 2007.
- [19] S. Somasundaran and J. Wiebe, "Finding the sources and targets of subjective expressions," *Proceedings of LREC*, 2008.
- [20] K. Nigam, "Retrieving topical sentiments from online document collections," *Document Recognition and Retrieval XI*, 2004.
- [21] L. Barbosa, "Robust Sentiment Detection on Twitter from Biased and Noisy Data," *research.att.com*.
- [22] A. Go and R. Bhayani, "Twitter sentiment classification using distant supervision," *CS224N Project Report*, 2009.
- [23] A. Bermingham, "Classifying sentiment in microblogs: is brevity an advantage?" *Proceedings of the 19th ACM ...*, 2010.
- [24] P. Ekman, "The repertoire of nonverbal behavior: Categories, origins, usage, and coding," *Semiotica*, 1969.
- [25] R. Plutchik, "The nature of emotions," *American Scientist*, 2001.
- [26] C. Yang, K. H.-Y. Lin, and H.-H. Chen, "Building emotion lexicon from weblog corpora," pp. 133--136, Jun. 2007.
- [27] C. Alm and D. Roth, "Emotions from text: machine learning for text-based emotion prediction," ... *of the conference on Human Language ...*, 2005.
- [28] A. Popescu. (2010, February) Cassandra @ twitter: An interview with ryan king. [Online]. Available: <http://nosql.mypopescu.com/post/407159447/cassandra-twitter-an-interview-with-ryan-king>
- [29] J. Moore. (2011, March) Big data @ foursquare. [Online]. Available: <http://engineering.foursquare.com/2011/03/24/big-data-foursquare-slides-from-our-recent-talk/>

- [30] M. Kennedy. (2010, April) Mongoddb vs. sql server 2008 performance showdown. [Online]. Available: <http://www.michaelckennedy.net/blog/2010/04/29/MongoDBVsSQLServer2008PerformanceShowdown.aspx>
- [31] C. Chang, ``LIBSVM: a library for support vector machines," 2001.
- [32] M. Marcus and M. Marcinkiewicz, ``Building a large annotated corpus of English: The Penn Treebank," *Computational . . .*, 1993.

PART IV | APPENDIX



TABLES AND FIGURES

A.1 Background

Table A.1: The University of Pennsylvania (Penn) tagset, as proposed by Marcus et al. (32)

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
VCN	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


Table A.2: Regular expressions for matching features

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

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


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

Figure A.1: Screenshot of our labeller's search form



A screenshot of a search form. It consists of a text input field containing the text "e.g. nhs reform, edl, david cameron, unemployment" and a grey button labeled "search".

Figure A.2: Screenshot of labeller layout for annotating statuses



A screenshot of a web interface for annotating statuses. On the left, there is a yellow highlighted text area containing the text: "Cameron's NHS speech does nothing to address criticisms of health professionals I Left Foot Forward - http://j.mp/iWH8QH" followed by "fetched 12 Jun" and a blue "delete" link. To the right of this text area are two columns of form elements. The first column has a dropdown menu with "unsure" selected, a text input field with "subject e.g. nhs", and another dropdown menu with "unsure" selected. The second column has a text input field with "+ve phrases", a text input field with "-ve phrases", and a text input field with "e.g. anger". To the right of these input fields is a text input field with "sentiment phrases" and a grey button with a checkmark.

B

CODE EXAMPLES

B.1 Content retrieval

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

```
1 class Status
2   include Mongomapper::Document
3
4   # Attributes
5   key :text, String
6   key :source, String
7   key :source_id, Int
8   key :posted_at, DateTime
9   key :from, String
10
11  # Relationship attributes
12  key :classified_status, ClassifiedStatus
13  key :trained_status, TrainedStatus
14 end
```

Listing B.2: Example Twitter search API results

```
1 {
2   "results": [
3     {
4       "text": "@twitterapi, look at my example tweet!",
5       "to_user_id": 396524,
6       "to_user": "TwitterAPI",
7       "from_user": "jkoum",
8       "metadata":
9       {
10        "result_type": "popular",
11        "recent_retweets": 100
12      }
13     }
14   ]
15 }
```

```

12     },
13     "id":1478555574,
14     "from_user_id":1833773,
15     "iso_language_code":"nl",
16     "profile_image_url":"http://twitter.com/image.jpg",
17     "created_at":"Wed, 08 Apr 2009 19:22:10 +0000"
18   }
19 ],
20 "since_id":0,
21 "max_id":1480307926,
22 "refresh_url":"?since_id=1480307926&q=%40twitterapi",
23 "results_per_page":15,
24 "next_page":"?page=2&max_id=1480307926&q=%40twitterapi",
25 "completed_in":0.031704,
26 "page":1,
27 "query":"%40twitterapi"
28 }

```

B.2 Polarity classification

Listing B.3: Code for finding polarised clues, including negation handling

```

1 def polarised_clues
2   polarised = {:positive => [], :negative => []}
3   negate = false
4   self.parts_of_speech.each do |p|
5     negate = !negate if ["not", "n't", "never"].include? p["word"]
6     negate = false if ["pp", "ppc"].include? p["tag"]
7     positive = Core::ClueFinder.is_positive_clue? p["word"], p["tag"]
8     negative = Core::ClueFinder.is_negative_clue? p["word"], p["tag"]
9     polarised[:positive] << p if (!negate and positive) or (negate and
        negative)
10    polarised[:negative] << p if (!negate and negative) or (negate and
        positive)
11  end
12  return polarised
13 end

```

Listing B.4: Status object method for extracting patterns

```

1 def patterns

```

```

2  rules = [
3    ["jj", ["nn","nns"], nil],
4    ["rb","rbr","rbs"], ["jj"], ["nn","nns"]],
5    ["jj"], ["jj"], ["nn","nns"]],
6    ["nn","nns"], ["jj"], ["nn","nns"]],
7    ["rb","rbr","rbs"], ["vb","vbd","vbn","vbg"], nil]
8  ]
9
10 pos = self.parts_of_speech
11 trigrams = (pos + [nil]).each_cons(3).to_a
12
13 patterns = trigrams.select do |first, second, third|
14   rules.map do |rule|
15     rule[0].include?(first["tag"]) and
16     rule[1].include?(second["tag"]) and
17     (
18       !rule[2] or
19       (!third.nil? and !rule[2].include?(third["tag"]))
20     )
21   end.inject(false){|m,n| m or n}
22 end
23
24 return patterns
25 end

```

B.3 Topic extraction

Listing B.5: TopicExtraction method for extracting a topic's pattern from a status' part of speech tags

```

1  def extract_pattern topic, pos
2    topic_parts = @tagger.split_words topic
3    # create our n-grams
4    status_parts = pos.each_cons(phrase_parts.length).to_a
5    patterns = []
6    status_parts.each_index do |i|
7      part = status_parts[i]
8      if part.map{|p| p["word"]} == topic_parts
9        pattern = []
10       pattern << i>0 ? status_parts[i-1][-1]["tag"] : nil
11       pattern += part.map{|p| p["tag"]}

```

```

12     pattern << i<status_parts.length-1 ? status_parts[i+1][0]["tag"]
      : nil
13     patterns << pattern
14   end
15 end
16 return patterns
17 end

```

Listing B.6: TopicExtraction method for extracting a status' topic

```

1 def extract_topics status
2   n = @@patterns.map{|s| s.length}.max
3   grams = (3..n).map{|i| status.parts_of_speech.each_cons(i).to_a}
4   grams = grams.sum
5   grams = grams.select do |gram|
6     @@patterns.include? gram.map{|pos| pos["tag"]}
7   end
8   grams[1..-2].map{|pos| pos["word"]}
9 end

```
