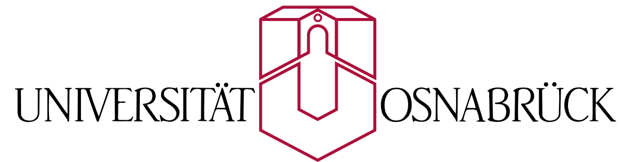

Institut für Informatik

Masterarbeit

Twitter Sentiment Analysis: On Feature Engineering, Classifier Performance and Realtime Tracking

Nils Haldenwang

September 2013

Erstgutachter: Prof. Dr. Oliver Vornberger
Zweitgutachterin: Prof. Dr. Elke Pulvermüller

Danksagungen

Hiermit möchte ich allen Personen danken, die mich bei der Erstellung der Arbeit unterstützt haben:

- Herrn Prof. Dr. Oliver Vornberger für die Tätigkeit als Erstgutachter und für die Bereitstellung der interessanten Thematik.
- Frau Prof. Dr. Elke Pulvermüller, die sich als Zweitgutachterin zur Verfügung gestellt hat.
- Frau Maren Mikulla, Frau Jana Lehnfeld und Herrn Nicolas Neubauer für das wertvolle Feedback während der Verfassung der Arbeit.
- Christian Benz, Marco Geertsema, Malte Gegner, Christian Heiden, Niels Hellwig, Rene Helmke, Steven Jones, Johannes Kerkloh, Christoph Knauft, Henning Krömker, Thorsten Langemeyer, Petr Legkov, Dominik Lips, Alexander Löhr, Michel Löpmeier, Isaak Mitschke, Konstantin Obermann, Jonas Rothe, Ronja-Verena Uder und Swen Wenzel für die Mithilfe bei der Bewertung der Tweets für das Testset.

Schließlich möchte ich insbesondere auch noch ganz herzlich meinen Eltern Heide und Edmund Haldenwang dafür danken, dass sie mir dieses Studium überhaupt ermöglicht haben.

Abstract

Millions of people publish their opinions about a variety of topics on the microblogging platform Twitter every day. Analyzing this stream of opinions automatically can be useful in various ways. For example, as a customer one might like to get an idea of the general opinion about a product before buying it. Another example are politicians who may be interested in the sentiment towards their political party, especially shortly before an election. As a first step towards such an automated analysis, algorithms which are able to determine the sentiment of a tweet are needed. This thesis provides a high quality testset to evaluate such algorithms, analyzes and compares various methods to classify a tweet's sentiment, and finally illustrates how a web based realtime sentiment tracking application, which tracks the sentiment towards given keywords, can be implemented.

Zusammenfassung

Auf der Microblogging-Plattform Twitter geben Millionen von Menschen ihre Meinung zu vielerlei Dingen in Form von Kurznachrichten preis. Diese Informationen können auf vielfältige Art genutzt werden. Als Privatperson ist man vielleicht vor dem Kauf eines Produktes an der öffentlichen Meinung bezüglich des Produktes interessiert. Auch Politiker möchten vor einer Wahl gerne wissen wie sie im Vergleich zur Konkurrenz dastehen. Zur Auswertung der großen Datenmengen sind Algorithmen notwendig, welche automatisiert die Stimmung eines Tweets erkennen können. Im Rahmen dieser Arbeit wird ein qualitativ hochwertiges Testset hinreichender Größe erstellt, mit dem eine Analyse und ein Vergleich verschiedener Klassifikationsmethoden durchgeführt werden kann. Schließlich wird dargestellt, wie eine Webapplikation zur Echtzeitverfolgung der Stimmung bezüglich gegebener Schlüsselworte umgesetzt werden kann.

Contents

1 Introduction **1**

1.1 Motivation 2

1.2 Objectives and Structure of the Thesis 3

2 Basics and State of the Art **5**

2.1 Naive Bayes Classification 5

2.1.1 Bayes Theorem 5

2.1.2 Classification 6

2.1.3 Dealing with Unknown Features: Smoothing Techniques 8

2.1.4 Implementation 10

2.2 Support Vector Machines 11

2.2.1 Origins and Basic Idea 12

2.2.2 Linear Hard-Margin SVMs 13

2.2.3 Soft-Margin SVMs 14

2.2.4 Non-Linear SVMs 15

2.2.5 Implementation 16

2.3 Representing Texts as Vectors 17

2.3.1 Word Level N-Grams 17

2.3.2 Sub-Word Level N-Grams 19

2.3.3 Preprocessing 19

2.3.4 Part-of-Speech Tagging 21

2.4 Overview of Current Research 25

3 Performance Investigation **29**

3.1 Construction of a General Purpose, High Quality Dataset 29

3.1.1 Quality Criteria 29

3.1.2 Labeling the data 30

3.1.3 Test Dataset Statistics 33

3.1.4 Collection and Analysis of Training Data 36

3.2 Measuring and Comparing Performance of Classifiers 39

3.3 Determining Training Corpus Size per Feature and Classifier 41

3.4 Effects of Preprocessing 46

3.5 Combining Features 47

3.5.1 Naive Bayes Classifier 47

3.5.2 Support Vector Machine 49

3.6 Conclusions 50

4 Implementation of a Real Time Sentiment Tracking Application **53**

4.1 Requirements 53

4.2 Overview of the Architecture and the Tools Used 54

4.3 Entity Management . . . 56
4.4 Harvesting and Processing Tweets . . . 58
4.5 Browsing an Entity's Tweets with Full-Text Search . . . 61
4.5.1 Presenting Tweets with Datatables . . . 61
4.5.2 Indexing and Retrieving Tweets with Elastic Search . . . 63
4.5.3 Connecting Datatables and Elastic Search Using the Presenter Pattern . . . 66
4.6 Visualizing the Entities' Sentiment . . . 68
4.6.1 Drawing Charts with Highcharts . . . 68
4.6.2 Computing Time Series Data with Facet Searches . . . 72
4.7 Conclusions . . . 74

5 Reflexion 75
5.1 Summary and Transferability of Results . . . 75
5.2 Conclusion and Outlook . . . 76

Bibliography 78

1 Introduction

Due to the enormous increase in web technologies and the rise of the Web 2.0, social media and micro blogs are among the most popular forms of communication these days. Even users of classical communication tools, such as mailing lists or blogs, tend to shift to microblogging platforms due to the easy accessibility and the ease of use compared to the traditional tools (Pak and Paroubek 2010). Every day, millions of messages are posted on microblogging websites like Facebook[1], Tumblr[2] and Twitter[3]. The messages cover a multitude of topics: Authors may write about their life, share opinions regarding various topics, such as products or politics, or just discuss current events and issues. Hence, microblogging tends to be a valuable source of people's opinions and sentiments which can be efficiently used for marketing or social studies.

For the research in this thesis Twitter has been chosen as the microblogging platform to be investigated. Messages on Twitter are called **tweets**. Figure 1.1 shows an example. A tweet's length cannot exceed 140 characters. If a tweet is considered particularly interesting, it can be **retweeted**, which means reposted by another person, similar to a quotation. Moreover, a tweet can contain special tokens, the so called **mentions** being one kind of tokens. A mention begins with a @-character, followed by the name of a user, for example *@NilsHaldenwang*. Using a mention results in the mentioned user being notified about it. Another kind of special token is the **hash tag**. Hash tags start with a #-char, followed by a keyword indicating the topic of the tweet, for example *#android*.

Figure 1.1: Example screen shot of a tweet with positive sentiment.

The reasons to choose Twitter from all available microblogging platforms are similar to those of Pak and Paroubek (2010) and Bakliwal et al. (2012). In 2012, Twitter had 465 million users which produced 175 million messages a day[4]. Thus, one can collect an arbitrary large corpus easily with the help of the provided API[5]. Due to the character limit of 140 characters tweets are considered to be less ambiguous than other messages.

[1] http://www.facebook.com
[2] http://www.tumblr.com
[3] http://www.twitter.com
[4] http://blog.sironaconsulting.com/.a/6a00d8341c761a53ef016767bafa2c970b-pi
[5] https://dev.twitter.com

Moreover, Twitter's user base is made up of people from various socio-cultural domains. There are lots of regular users but also celebrities, company representatives and even politicians who use Twitter. Furthermore, the messages are written by authors from multiple countries, the majority of them (107.7 million of the 175 million per day) coming from the United States. For this reason, this work is focussed on the analysis of tweets written in the English language.

1.1 Motivation

Twitter offers an arbitrarily large amount of opinions and attitudes towards numerous topics, for example products, politics, celebrities and many more. Some say it can be considered as an "Electronic Word of Mouth" (Jansen et al. 2009). Monitoring and analyzing this data provides enormous opportunities for both public and private sectors. Observations indicate a strong correlation between rumours and negative opinions, which have been shared by users on social networks, and the reputation of a certain product or company (Saif et al. 2012a, Ward and Ostrom 2003, Yoon et al. 1993). Therefore, the consideration of microblogging platforms like Twitter can help companies to improve their relationship with the customers, understanding their customers' needs and reacting better to changes in the market (Saif et al. 2012a).

The results of Asur and Huberman (2010) strengthen this hypothesis. They found a strong correlation between the rate of tweets with positive sentiment and the box-office revenue of movies. Especially interesting is the fact that a change of polarity towards a movie before and after release has a strong influence on the box-office revenue. To measure the polarity of tweets about a movie, the following polarity ratio has been introduced:

$$PNratio = \frac{|\text{tweets with positive sentiment}|}{|\text{tweets with negative sentiment}|} \tag{1.1}$$

The movie *New Moon*, for example, started out with a polarity ratio of 6.29 and a box-office revenue of 142M in the first week. Due to a downfall of the polarity ratio to 5 in the second week, the box-office revenue also dropped to 42M. On the contrary, increasing box-office revenues were recognized in conjunction with an uprise of polarity. The movie *The Blind Side* started out with a polarity ratio of 5.02 and opening week sales of 34M. However, in the second week the polarity ratio increased to 9.65, which led to revenues of 40.1M.

Taking into account that the sentiment of tweets influences the reputation of products and companies, they also may correlate with stock prices. The work of Bollen et al. (2011b) revealed a strong correlation between the public mood at Twitter and the Dow Jones Industrial Average (DIJA). Thus, they were able to significantly improve the DIJA closing value prediction of a Self-Organizing Fuzzy Neural Network by adding the mood state of the public Twitter stream as additional inputs. Moreover, these results have been verified recently by Mittal and Goel (2012).

The results of Bollen et al. (2011a) suggest that the public mood represented by the Twitter stream responds strongly to political or cultural events like the U.S. Presidential Election of November 4, 2008 and Thanksgiving Day. Because of this they propose the usage of the public mood to detect such events which may not be as obvious as an election or a holiday. Another use case for the socio-economic domain is the realtime analysis of political debates. The first U.S. presidential debate in 2008 was analyzed by Diakopoulos and Shamma (2010). They have been able to identify key sections of the debate by looking at the polarity of related tweets. In addition, they tracked the sentiment towards the participants (Obama and McCain) over time and found Obama to be more popular.

It has been shown that the reliable classification of a tweet's sentiment has many real world use cases. Therefore, it is of great importance to evaluate and compare current methods to get a deeper insight into their strengths and weaknesses to further improve their real world applications.

1.2 Objectives and Structure of the Thesis

This thesis is aimed towards three major objectives. Firstly, a dataset of high quality shall be created to be able to evaluate and compare various methods for Twitter Sentiment Analysis. Secondly, standard classifiers, features and preprocessing techniques are looked at in detail to clear up contradictory claims made by current researchers. This should be done by evaluating them with the created testset. Thirdly, a general concept to apply the obtained classifier should be illustrated by implementing a realtime sentiment tracking application.

Chapter 2, *Basics and State of the Art*, first establishes the basic knowledge necessary to understand the current methods. This basics consist of describing two standard classification algorithms and introducing basic feature engineering techniques along with common preprocessing methods. The chapter concludes with an overview of the current research regarding Twitter Sentiment Analysis.

In chapter 3, *Performance Investigation*, the creation of a high quality test dataset is described. Moreover, this dataset is analyzed with respect to the features introduced in the preceding chapter. Additionally, it provides information about how performance of classifiers can be measured. Finally, a variety of methods and algorithms are evaluated with the created testset and the results are compared and discussed.

The implementation of the exemplary realtime sentiment tracking system is described in chapter 4, *Implementation of a Real Time Sentiment Tracking Application*. After defining the requirements and providing an overview of the architecture and the tools used, a more detailed description of each feature's implementation is given.

Chapter 5, *Reflexion*, concludes the thesis by summarizing the results, discussing their transferability to other domains, draws a final conclusion and provides an outlook on further work.

2 Basics and State of the Art

In this chapter basic knowledge necessary to understand the current methods for Twitter Sentiment Analysis is presented first. The aforementioned basics consist of the introduction of two standard classification methods, followed by an illustration of current feature engineering methodologies and data preprocessing techniques. Finally, an overview of the current methods is provided and their results are discussed.

2.1 Naive Bayes Classification

Bayesian classifiers are statistical classifiers which are able to predict the probability of a given sample to belong to a particular class. The simplest Bayesian classifier, known as **Naive Bayes Classifier** (NBC), is comparable in performance with **Decision Trees**, **Neural Networks** (Han et al. 2006) and, using various smoothing techniques (Yuan et al. 2012), even with **Support Vector Machines** (SVM).

2.1.1 Bayes Theorem

The following explanation is taken, but slightly simplified and shortened, from Han et al. (2006).

Let X be a data sample with unknown class label and let H be a hypothesis, such as that the data sample X belongs to a specified class C. The classification problem is to determine $P(H|X)$, the probability that given a sample X the hypothesis H holds.

$P(H|X)$ is called **posterior probability** of H conditioned on X. Within the domain of Twitter Sentiment Analysis the data samples consist of tweets, their canonical features being the words. Given X contains the words *sad* and *bad* and H is the hypothesis that the tweet has a negative sentiment. Then $P(H|X)$ reflects the confidence that X has a negative sentiment, given we know it includes the words *sad* and *bad*.

In contrast, $P(H)$ is the **prior probability** of H, the probability of the hypothesis holding for any given sample X. For example, this is the probability that any given tweet has a negative sentiment, regardless of which words it contains. Note the prior probabilities independence of X, whereas the posterior probability is based on additional information (such as background knowledge).

Similarly, $P(X|H)$ is the posterior probability of X conditioned on H. In the example this would be the probability that X contains the words *sad* and *bad*, given we know its sentiment is negative. $P(X)$ is the prior probability of X, in the example it is the probability of a tweet containing the words *sad* and *bad*.

The question is: How can these probabilities be estimated? First of all, $P(X)$, $P(H)$ and $P(X|H)$ may be estimated from the given data. To finally calculate the posterior probability from these probabilities one can harness the **Bayes Theorem**:

$$P(H|X) = \frac{P(X|H)P(H)}{P(X)} \ . \tag{2.1}$$

2.1.2 Classification

This section is also based on Han et al. (2006) and slightly adapted to fit the needs of this thesis.

Each data sample is represented by an n-dimensional vector $X = (x_1, x_2, \ldots, x_n)$, depicting n measurements made on the sample from n attributes $A_1, A_2, \ldots, A_n$.

Let $C_1, C_2, \ldots, C_m$ be m classes to which an unknown given sample X can be assigned. The classifier will predict X to belong to the class C_i having the highest posterior probability, conditioned on X. That is, the naive Bayes classifier assigns an unknown sample X to class C_i if and only if

$$P(C_i|X) > P(C_j|X) \text{ for } 1 \leq j \leq m, \ j \neq i \ . \tag{2.2}$$

Thus, we maximize $P(C_i|X)$. The class C_i for which $P(C_i|X)$ is maximized is called the **maximum posteriori hypothesis**. It can be computed using the Bayes Theorem 2.1:

$$P(C_i|X) = \frac{P(X|C_i)P(C_i)}{P(X)} \ . \tag{2.3}$$

Owing to the fact that $P(X)$ is constant for all classes, $P(X|C_i)P(C_i)$ needs to be maximized. One problem which can occur is a lack of knowledge about the class prior probabilities. Therefore, it is commonly assumed that the classes' occurrences are equally likely: $P(C_1) = P(C_2) = \cdots = P(C_m)$. One would maximize $P(X|C_i)$. Otherwise $P(X|C_i)P(C_i)$ would be maximized. Nevertheless, the class prior probability may be estimated by

$$P(C_i) = \frac{s_i}{s} \ , \tag{2.4}$$

where s_i is the number of training samples of class C_i and s is the total number of training samples.

Text data can contain a nearly infinite number of attributes because there is no limit for forming words, especially due to the excessive use of slang and abbreviations on microblogging platforms. Thus, it would be extremely computational expensive, or even impossible, to compute $P(X|C_i)$. Consequently, the computation has to be reduced somehow. Simplifying the computation can be done by making the naive assumption

of **class conditional independence**. This assumption presumes that the values of the attributes are conditionally independent of one another, given the class label of the sample. That is, there are no dependence relationships among the attributes. Thus, $P(X|C_i)$ can be computed like this:

$$P(X|C_i) = \prod_{k=1}^{n} P(x_k|C_i) \ . \tag{2.5}$$

The computation of $P(x_k|C_i)$ can be done with the **maximum likelihood estimation**

$$P(x_k|C_i) = \frac{s_{ik}}{s_i} \ , \tag{2.6}$$

where s_{ik} is the number of training samples of class C_i having the value x_k for A_k, and s_i is the number of training samples belonging to C_i.

When dealing with text data, especially short texts like tweets, one would not consider the number of occurrences of a word, but just its presence. Thus, the maximum likelihood estimation is often (Joachims 2002, Saif et al. 2012a;b) formulated as

$$P(w|C_i) = \frac{TF(w, C_i)}{\sum_{w' \in V} TF(w', C_i)} \ , \tag{2.7}$$

where $TF(w, C_i)$ is the occurrence frequency of word w in documents of class C_i, and V is the vocabulary of the underlying text corpus.

In order to classify an unknown sample X, $P(X|C_i)P(C_i)$ is evaluated for each class C_i. Sample X is then assigned to the class C_i if and only if

$$P(X|C_i)P(C_i) > P(X|C_j)P(C_j) \text{ for } 1 \leq j \leq m, \ j \neq i \ . \tag{2.8}$$

Simply put, it is assigned to the class C_i for which $P(X|C_i)P(C_i))$ is the maximum:

$$\text{classify}(X) = \arg\max_{C_i} P(X|C_i)P(C_i) \ . \tag{2.9}$$

For text classification problems, using equations 2.5 and 2.7, the classification function can be formulated as:

$$\text{classify}(X) = \arg\max_{C_i} \ P(C_i) \prod_{k=1}^{n} \frac{TF(w_k, C_i)}{\sum_{w' \in V} TF(w', C_i)} \ , \tag{2.10}$$

with n being the number of words in X.

Since the classification function is basically a product of many small numbers, the probability is often transformed to the so called **log likelihood**, to reduce floating point errors while computing the results (Pak and Paroubek 2010, Bonev et al. 2012):

$$\text{classify}(X) = \arg\max_{C_i} \left(\log P(C_i) \; + \; \sum_{k=1}^{n} \log \frac{TF(w_k, C_i)}{\sum_{w' \in V} TF(w', C_i)} \right) \; . \tag{2.11}$$

2.1.3 Dealing with Unknown Features: Smoothing Techniques

When the language is informal, such as that in tweets, there are many unknown words, which are not covered by training data. Furthermore, the maximum likelihood estimation for $P(w|C_i)$ (see equation 2.7) cannot be computed because the divisor would be zero. Hence, the model would assign unknown words a zero probability for all of the classes, which is probably not true. Various smoothing techniques have been introduced to deal with this problem by estimating a probability for unknown words.

Zhai and Lafferty (2004) summarize: "In general, smoothing methods discount the probabilities of words seen in the text and assign the extra probability mass to the unseen words according to some fallback model." As their field of research was information retrieval, they exploited the collection language model as fallback. For the purpose of twitter sentiment analysis, the fallback model may be exchanged, but for now the explanation of the general principle will stick to the definitions of Zhai and Lafferty (2004). The nomenclature is slightly adapted to fit the previous explanations.

Chen and Goodman (1996) assume the general form of a smoothed model to be the following:

$$P(w|C_i) = \begin{cases} P_s(w|C_i), & \text{if word } w \text{ is seen} \\ \alpha_d P(w|M), & \text{otherwise} \end{cases} \; . \tag{2.12}$$

In this equation $P_s(w|C_i)$ is the smoothed probability of a seen word, $P(w|M)$ is the collection language model and α_d is a coefficient controlling the probability mass assigned to unseen words, so that all probabilities sum up to one. Given $P_s(w|C_i)$, α_d must have the form:

$$\alpha_d = \frac{1 - \sum_{w \in V: TF(w, C_i) > 0} P_s(w|C_i)}{1 - \sum_{w \in V: TF(w, C_i) > 0} P(w|M)} \; . \tag{2.13}$$

Hence, the essential difference of smoothing methods is the choice of $P_s(w|C_i)$.

The easiest smoothing method coming to mind is called **Laplace smoothing**. It was suggested by Vapnik (1982), and its idea is as simple as adding an extra count to every word. Even though the idea is not complicated, this technique works well in

practice (Joachims 1996; 2002). Applied to the maximum likelihood estimation for text classification (see equation 2.7), the new estimator looks like this:

$$P_L(w|C_i) = \frac{1 + TF(w, C_i)}{|V| + \sum_{w' \in V} TF(w', C_i)} \ . \tag{2.14}$$

Yuan et al. (2012) argue that just adding one to the occurrence frequency of the word just adds noise, to which classes containing few training data samples are very sensitive. However, as this is not the case for twitter sentiment classification, Laplace smoothing can still be used. For instance, it was used as baseline in conjunction with an unigram language model by Saif et al. (2012b;a).

Another smoothing technique is the **Jelinek-Mercer method** introduced by Jelinek and Mercer (1980). The maximum likelihood model is linearly interpolated with the fallback model. A coefficient λ is used to control the influence of each:

$$P_\lambda(w|C_i) = (1 - \lambda) \frac{TF(w, C_i)}{\sum_{w' \in V} TF(w', C_i)} + \lambda P(w|M) \ . \tag{2.15}$$

This is basically a simple mixture model to mingle two distributions with a given weight. This smoothing technique also has been used for twitter sentiment classification with various fallback models (Saif et al. 2012b;a, Liu et al. 2012).

One may also consider **Bayesian smoothing using Dirichlet priors**. According to MacKay and Peto (1995) a language model is a multinomial distribution, for which the conjugate prior for Bayesian analysis is the Dirichlet distribution. Zhai and Lafferty (2004) choose the parameters of the Dirichlet to be:

$$(\mu P(w_1|M),\ \mu P(w_2|M),\ \ldots,\ \mu P(w_n|M)) \ . \tag{2.16}$$

Therefore, the model is given by:

$$P_\mu(w|C_i) = \frac{TF(w, C_i) + \mu P(w|M)}{\sum_{w' \in V} TF(w', C_i) + \mu} \ . \tag{2.17}$$

It may be noticed that the Laplace method is just a special case of Bayesian smoothing using Dirichlet priors, with $P(w|M) = \frac{1}{|V|}$ and $\mu = |V|$. To the best of my knowledge there are no further applications to twitter sentiment analysis except the afore mentioned ones using the Laplace method.

In addition, there is the **Absolute discounting**. To lower the probability of seen words, a constant is subtracted from the word's count (Zhai and Lafferty 2004). This method is similar to the Jelinek-Mercer method, the difference being subtraction of a constant instead of multiplication with $(1 - \lambda)$:

$$P_\delta(w|C_i) = \frac{max(TF(w, C_i) - \delta, 0)}{\sum_{w' \in V} TF(w', C_i)} - \sigma P(w|M) \ , \tag{2.18}$$

where $\delta \in [0,1]$ is the discount constant and $\sigma = \frac{\delta |C_i|_u}{|C_i|}$, assuring all probabilities sum to one. The term $|C_i|_u$ denotes the number of *unique* terms in class C_i, whereas $|C_i|$ is the total count. This method, to the best of my knowledge, has neither been used for twitter sentiment classification so far.

Finally, it can be beneficial to incorporate multiple smoothing methods, one example being **Two-stage smoothing** (Yuan et al. 2012):

$$P_{\lambda,\mu}(w|C_i) = (1-\lambda)\frac{TF(w,C_i) + \mu P(w|M)}{\sum_{w' \in V} TF(w',C_i) + \mu} + \lambda P(w|M) \ . \qquad (2.19)$$

This example combines the Jelinek-Mercer method with Bayesian Smooting using Dirichlet priors. Yuan et al. (2012) report this method to perform reasonably well for topic classification of short questions, which is very similar to sentiment classification of tweets.

All in all, there are multiple smoothing methods available, of which some have been used for twitter sentiment classification or similar tasks. However, it still has to be investigated which of those methods yields the best results in conjunction with various features.

2.1.4 Implementation

Due to the simplicity of the computations it is not necessary to use any special framework. Naive Bayes Classifiers could be easily implemented in very few lines of code.

Listing 1 shows a simple example implementation of a Naive Bayes Classifier for text-classification using unigram features (see section 2.3.1 for details) with Laplace smoothing. For reasons of simplicity other smoothing methods are left out here but will be used in the evaluation of course.

The constructor `initialize` creates a `Hash` which stores the term frequencies for the classes in another nested `Hash`. Furthermore, it creates another `Hash` to store the total term frequencies of all words in the training corpus. To train the classifier with examples, the method `train` is used, which expects a training sample of type `String` and its corresponding class label as parameters. For each word in the sample it increments the term frequency for the correct class and the total term frequency. The method `maximum_likelihood_estimation` computes the maximum likelihood estimation from the learned language model for a given term and class, according to equation 2.7. Finally, the method `classify` can be used to classify an unknown sample. It selects the class with the highest probability, according to 2.11, leaving out the prior probabilities for the classes, since in general they are not known for most text-classification problems.

```ruby
class NaiveBayesClassifier
  def initialize
    @class_term_frequencies = Hash.new { Hash.new(0) }
    @total_term_frequencies = Hash.new(0)
  end

  def train(example, class_label)
    example.terms.uniq.each do |term|
      @class_term_frequencies[class_label][term] += 1
      @total_term_frequencies[term] += 1
    end
  end

  def maximum_likelihood_estimation(term, klass)
    (@class_term_frequencies[klass][term] + 1.0) /
      (@total_term_frequencies[term] + @total_term_frequencies.
       inject(0) { |k, v, tmp| tmp + v })
  end

  def classify(sample)
    result = Hash.new(0)
    sample.each do |term|
      @class_term_frequencies.each_key do |klass|
        result[klass] += Math.log(
            maximum_likelihood_estimation(word, klass)
        )
      end
    end
    result.keys.sort_by {|key| result[key]}.last
  end
end
```

Listing 1: Naive Bayes Classifier written in the programming language Ruby.

2.2 Support Vector Machines

This section gives an introduction to **Support Vector Machines** (SVMs), based on Joachims (2002, chapter 3). SVMs are non-probabilistic linear binary classifiers, which can be used for classification and regression analysis. They are able to handle large feature spaces reasonably well. Firstly, the history of SVMs will be highlighted shortly, followed by an illustration of the various types of SVMs. While the standard SVM is a linear classifier, non-linear problems also can be handled using a so called **kernel trick**, which is also illustrated. The chapter concludes with a discussion on currently available SVM implementations.

2.2.1 Origins and Basic Idea

Support Vector Machines were developed based on the *Structural Risk Minimization* principle (Vapnik 1982, Cortes and Vapnik 1995, Vapnik 1998). The idea is to find a hypothesis h from a hypothesis space H, for which the lowest error probability $Err(h)$ can be guaranteed for a given sample S:

$$(\vec{x}_1, y_1), \ldots, (\vec{x}_n, y_n) \qquad \vec{x}_i \in \mathbb{R}^n,\ y_i \in [-1, +1]\ , \tag{2.20}$$

where $\vec{x}_i$ denotes a feature vector and y_i the class label. The true error of a hypothesis h is connected with the error $Err_{train}(h)$ of h on the training set and the complexity of h by the following upper bound (Vapnik 1998):

$$Err(h) \leq Err_{train}(h) + O\left(\frac{d\ ln(\frac{n}{d}) - ln(\eta)}{n}\right)\ . \tag{2.21}$$

The probability of the bound holding is at least $1 - \eta$. Furthermore, d denotes the so-called **VC-dimension** (Vapnik 1998), which indicates the expressiveness of the hypothesis space H. Equation 2.21 reflects a trade-off between complexity of the hypothesis space and the training error. On the one hand, a simple hypothesis space with a small VC-dimension will probably not contain good approximation functions. Thus, the training error, along with the true error, will be large. On the other hand, a very large hypothesis space (large VC-dimension) will lead to a smaller training error, but will also increase the upper bound due to its linear influence in the right hand side term of equation 2.21.

Therefore, when the hypothesis space has a high VC-dimension, the hypothesis with a very low training error may just fit the training data without proper generalization. This results in poor performance when predicting unknown examples. In general, such behavior of machine learning algorithms is called **overfitting**. Hence, it is crucial to pick a hypothesis space with correct complexity.

In Structural Risk Minimization, the prevention of overfitting is achieved by nesting hypothesis spaces H_i in a way that their respective VC-dimension d_i increases:

$$H_1 \subset H_2 \subset H_3 \subset \cdots \subset H_i \subset \ldots \quad \text{and} \quad \forall i : d_i \leq d_{i+1}\ . \tag{2.22}$$

This structure has to be defined before analyzing the training data. The problem to be solved is to find an index i^* for which the training error is minimal.

The question is: How can those structures be found in practice?

In Structural Risk Minimization, linear threshold functions with N features are created, resulting in the function's VC-complexity being $N + 1$. Given the features as a ranked list, using the first feature will have a VC-dimension of two, using the first two features will have a VC-dimension of three and so on. For very large feature spaces, as it is the case in text classification, this is not practical. Moreover, it is not clear how to rank the features.

Support Vector Machines learn linear threshold functions of the type:

$$h(\vec{x}) = sign\{\vec{w} \cdot \vec{x} + b\} = \begin{cases} +1, & \text{if } \vec{w} \cdot \vec{x} + b > 0 \\ -1, & \text{otherwise} \end{cases} \tag{2.23}$$

These functions correspond to a hyperplane within the feature space. This hyperplane is described by $\vec{w}$, being the hyperplanes normal vector, and b, being the offset from the origin along this normal vector. All vectors $\vec{x}$, which satisfy equation $\vec{w} \cdot \vec{x} + b = 0$, lie within the hyperplane. Hence, classification of an example $\vec{x}$ with $h(\vec{x})$ basically is done by determining on which side of the hyperplane it lies. Vapnik (1998) showed that the VC-dimension of Support Vector Machines is independent of the number of features, but is bound by the margin δ (see the following section 2.2.2 for an explanation of δ). Vapnik (1998) also showed that the VC-dimension becomes smaller the larger the margin δ is. While this property does not guarantee good performance, it guarantees that SVMs do not *necessarily* fail, meaning they are able to perform well for high dimensional classification tasks with a reasonable VC-dimension. For further details see Joachims (2002) and Vapnik (1998).

2.2.2 Linear Hard-Margin SVMs

Let the training samples be tuples $(\vec{x}_i, y_i)$ with $\vec{x}_i$ denoting the vector of feature values and $y_i \in \{-1, +1\}$ denoting the class labels. For simplicity it is assumed that the data is linearly separable, meaning it can be divided by at least one hyperplane h'. Thus, a weight vector $\vec{w}'$ and a threshold b' exist, such that all positive examples are on one side of the hyperplane, and the negative examples are on the other side. This is equivalent to:

$$\forall(\vec{x}_i, y_i): \quad y_i(\vec{w}' \cdot \vec{x}'_i + b') > 0 \ . \tag{2.24}$$

As shown in figure 2.1a there can be an arbitrarily large number of hyperplanes separating the classes without errors. From these the Support Vector Machine chooses the hyperplane h^* with the largest margin δ, as shown in figure 2.1b. Training samples closest to the hyperplane, the distance to it being exactly δ, are called **Support Vectors**. They are marked with circles.

To find the hyperplane h^* with maximum margin one has to solve the following optimization problem:

Optimization Problem 1 (Hard-Margin SVM (PRIMAL))

$$minimize: \quad V(\vec{w}, b) = \frac{1}{2}\vec{w} \cdot \vec{w} \tag{2.25}$$

$$subject\ to: \quad \forall_{i=1}^{n}: y_i(\vec{w} \cdot \vec{x}_i + b) \geq 1 \ . \tag{2.26}$$

Equation 2.26 formalizes the condition that every example has to be on the correct side of the hyperplane. Unlike in equation 2.24 the inequalities' right hand side is now one and not zero anymore. This enforces a certain margin δ. As the weight vector $\vec{w}$ also is

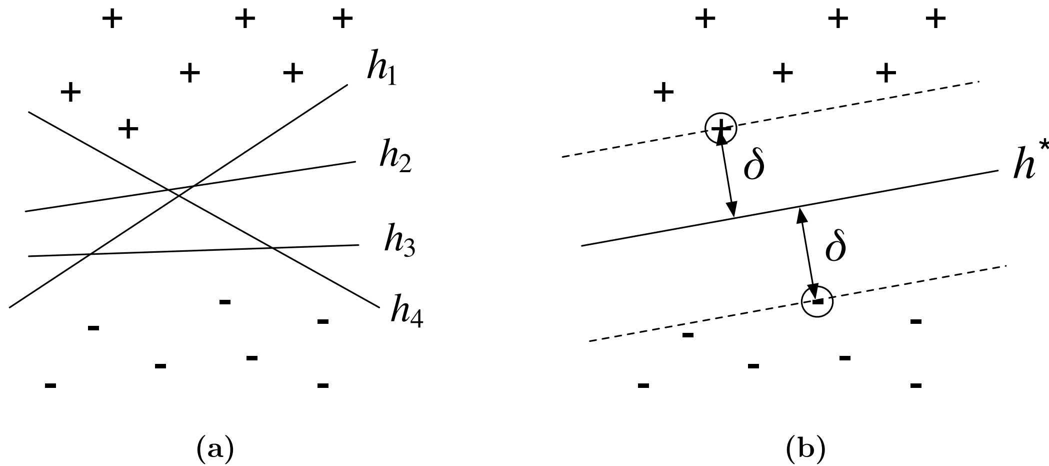

Figure 2.1: Example of a two dimensional binary classification problem. Positive examples are marked by + and negative ones by −. The left figure (a) shows that many hyperplanes separate the training samples without error. Support Vector Machines find the hyperplane h^*, which separates the training examples with maximum margin δ, as shown in the right figure (b). The examples closest to the hyperplane are called **support vectors** (marked with circles). Also see Joachims (2002).

the normal vector of the hyperplane, it is easy to verify that $\delta = \frac{1}{||\vec{w}||}$ with $||\vec{w}||$ being the L_2-norm of the vector $\vec{w}$. Hence, by minimizing $\vec{w} \cdot \vec{w}$ the margin δ is maximized. The hyperplane h^* is described by $\vec{w}$ and b, which are the solution of the optimization problem.

As this optimization problem is numerically hard to solve, it is often transformed to its Wolfe dual, an equivalent problem having the same solution, which is commonly solved in practice. For further details see Joachims (2002).

2.2.3 Soft-Margin SVMs

The Linear Hard-Margin SVM suffers from the disadvantage, that its training fails if the data is not linearly separable. In this case, there will be no feasible solution to Optimization Problem 1. Although most text-classification problems are linearly separable (Joachims 2002), it may still be beneficial to allow a certain number of errors in training. To overcome this issue, Cortes and Vapnik (1995) developed the **Soft-Margin SVM** by incorporating an upper bound to the number of training errors into Optimization Problem 1 and minimize it along with the weight vector:

Optimization Problem 2 (Soft-Margin SVM (PRIMAL))

$$\textit{minimize}: \quad V(\vec{w}, b) = \frac{1}{2}\vec{w} \cdot \vec{w} + C \sum_{i=1}^{n} \xi_i \tag{2.27}$$

$$\textit{subject to}: \quad \forall_{i=1}^{n} : y_i(\vec{w} \cdot \vec{x}_i + b) \geq 1 - \xi_i \tag{2.28}$$

$$\forall_{i=1}^{n} : \xi_i > 0 \tag{2.29}$$

The ξ_i are called **slack variables**. To satisfy condition 2.28, those have to be greater than one if the corresponding training sample lies on the wrong side of the hyperplane. Therefore, $\sum_{i=1}^{n} \xi_i$ is an upper bound for the number of training errors. Parameter C can be used to control how errors are tolerated. Large values for C lead to the Soft-Margin SVM to behave similar to a Hard-Margin SVM because even slack variables with small values lead to large increases of the objective functions value. Small values for C will lessen the influence of the slack variables and hence allow for more errors. Finally, condition 2.29 prevents the assignment of zero to all slack variables, as this would be always optimal but does not take any of the ξ_i into account. Following the strategy to solve Optimization Problem 1 for Hard-Margin SVMs, Optimization Problem 2 is also transformed to its Wolfe dual due to numerical problems when solving it directly. See Joachims (2002) for further information.

2.2.4 Non-Linear SVMs

The SVMs mentioned so far can only handle linear classification problems. Even though text-classification problems are claimed to usually be linearly separable (Joachims 2002, Fan et al. 2008), some of them, along with many other real world problems, are not linearly separable. Fortunately, Boser et al. (1992) developed a method which enables the possibility to easily transform SVMs to non-linear learners. The attribute vectors $\vec{x}_i$ are basically just mapped into a higher dimensional space X' with a non-linear mapping function $\Phi(\vec{x}_i)$. The SVM then learns the linear maximum margin method as before but in the new feature space X' of higher dimension, where the data is now linearly separable. Even though the learned classification rule is linear in X', it is non-linear when transformed back to the initial feature space.

The following example, taken from Joachims (2002), illustrates the afore mentioned transformation for two input variables x_1 and x_2. One chooses

$$\Phi((x_1, x_2)^T) = (x_1^2, x_2^2, \sqrt{2}x_1x_2, \sqrt{2}x_1, \sqrt{2}x_2, 1)^T \tag{2.30}$$

as a non-linear mapping function to transform the attribute vectors to X'. It is impossible to linearly separate the data as illustrated in the left-hand image (a) of figure 2.2. Yet, when mapping the data to another feature space using $\Phi(\vec{x})$, as shown in the right-hand side image (b) of 2.2, the data becomes linearly separable. One possible linear separator (although not with maximum margin) would be the weight vector $\vec{w} = (-1, 0, 0, 0, \sqrt{2}, 0)^T$ with $b = 0$ (it is illustrated as dotted line in both images of figure 2.2).

In general, the mapping function $\Phi(\vec{x})$ cannot be efficiently computed. Boser et al. (1992) have been able to solve this problem. They found it to be sufficient to compute the dot product $\Phi(\vec{x}_i) \cdot \Phi(\vec{x}_j)$ in the new feature space, when solving the dual optimization problems. For some special cases of $\Phi(\vec{x})$ those can be efficiently computed using so called **kernel functions** $\kappa(\vec{x}_1, \vec{x}_2)$. As long as those kernel functions satisfy Mercer's

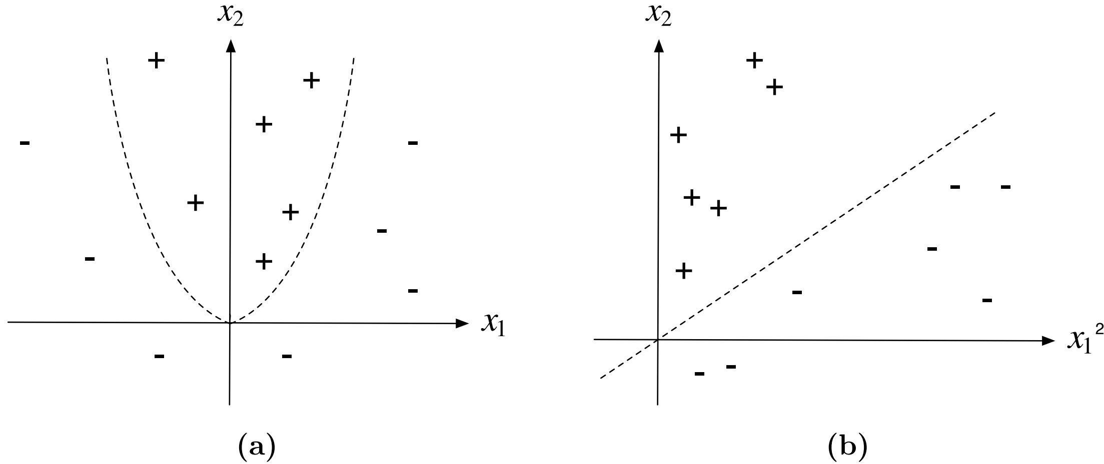

Figure 2.2: The training set shown in the left-hand graph (a) is obviously not linearly separable in (x_1, x_2). A non-linear transformation of the form (x_1^2, x_2) is depicted in the right-hand graph (b). Within this new space, the training examples are linearly separable. Also see Joachims (2002).

Theorem, they are guaranteed to compute the inner product of mapped vectors in the feature space X' (Vapnik 2000):

$$\kappa(\vec{x}_1, \vec{x}_2) = \Phi(\vec{x}_1) \cdot \Phi(\vec{x}_2) \ . \tag{2.31}$$

Depending on the choice of the kernel function, SVMs are able to learn polynomial classifiers, radial basis function (RBF) classifiers or two layer sigmoid neural networks:

$$\kappa_{\text{poly}}(\vec{x}_1, \vec{x}_2) = (\vec{x}_1 \cdot \vec{x}_2 + 1)^d \tag{2.32}$$

$$\kappa_{\text{rbf}}(\vec{x}_1, \vec{x}_2) = exp(-\gamma(\vec{x}_1 - \vec{x}_2)^2) \tag{2.33}$$

$$\kappa_{\text{sigmoid}}(\vec{x}_1, \vec{x}_2) = tanh(s(\vec{x}_1 \cdot \vec{x}_2) + c) \ . \tag{2.34}$$

The kernel function for the mapping of the example from above is $\kappa_{\text{poly}} = (\vec{x}_1 \cdot \vec{x}_2 + 1)^2$. This is obviously much more efficient than enumerating all possible polynomial terms, like in polynomial regression.

The incorporation of the kernel functions into the learning process is done by replacing every occurrence of inner products within the dual optimization problems with the chosen kernel function. See Joachims (2002) for further details.

2.2.5 Implementation

Due to the complexity of the training process and considering the fact that the main focus of this thesis is not the internals of SVMs, the classifier is not implemented from scratch.

Various SVM libraries have been released, one of the most popular being **LIBSVM** (Chang and Lin 2011). LIBSVM is well documented. Moreover, its efficiency has been proven in various contests[1]. All the aforementioned types of SVMs are implemented and can be used to evaluate the performance of various features and parameters.

As it is claimed that most text-classification problems are linearly separable (Joachims 2002, Fan et al. 2008) another library called **LIBLINEAR** was developed. It is optimized for linearly separable large scale problems with sparse attribute vectors. This is the case for almost all text-classification problems. A tweet, for example, contains only a few words, whereas the dimension of the input space is the number of all known words. Hence, the attribute vector of a tweet contains lots of zeros. Such a vector is called sparse. The authors of LIBLINEAR (Fan et al. 2008) claim that it can solve some of these problems in a few seconds, whereas it takes LIBSVM several hours to do so. Thus, LIBLINEAR provides a valuable alternative to evaluate linear SVMs and is used instead of LIBSVM in this thesis.

2.3 Representing Texts as Vectors

Both methods (NBC and SVM) require numerical feature vectors as inputs. In this section the most common practices of transforming a given text document into such a feature vector are introduced, and their relevancy for analyzing the sentiment of tweets is discussed.

2.3.1 Word Level N-Grams

The canonical way of representing texts as vectors is transforming them by utilizing so called **n-grams**. An n-gram is a contiguous sequence of n items from a given sequence of text. Table 2.1 shows the n-grams for $n \in \{1, 2, 3\}$ of the sentence *The sun is shining today.* Groups of n words are formed for word-level n-grams. The first group begins with the first word and also contains the following $n-1$ words. After this, the second group is constructed by starting with the second word and taking the following $n-1$ words. Finally, the described process is continued until the end of the sequence is reached. To put it simple one could say that a window of n words is constructed which moves over the text word by word and puts a snapshot of each position into to result.

Unigrams, $n = 1$	**Bigrams**, $n = 2$	**Trigrams**, $n = 3$
The, sun, is, shining, today	The sun, sun is, is shining, shining today	The sun is, sun is shining, is shining today

Table 2.1: Unigrams, bigrams and trigrams for the sentence *The sun is shining today.*, separated by commas.

[1] http://clopinet.com/isabelle/Projects/NIPS2003/, http://www.causality.inf.ethz.ch/home.php, and http://www.causality.inf.ethz.ch/activelearning.php

Transforming a text into n-grams divides it into an ordered list of tokens which is not a vector yet. To construct a vector, each token is considered as a feature, which means each token is mapped to an index of the vector. Given the underlying text corpus' vocabulary V, the size of the vector will be $|V|$. Considering unigrams this could be tens to hundreds of thousands, as for a reasonably large corpus the number of words will be approximately the number of words in the language of the texts.

Finally, the values of the features have to be computed. These values are often set to the term frequency $TF(w, d)$ which is simply the number of occurrences of the word w in the document d. This model is also referred to as *bag-of-word* model. Figure 2.3 illustrates this. However, it is often claimed that it is beneficial to just use binary values for term presence in a document instead of its frequency (Pak and Paroubek 2010). There is usually no difference in these two approaches regarding tweets, since due to their short lengths the words do not occur more then once anyway.

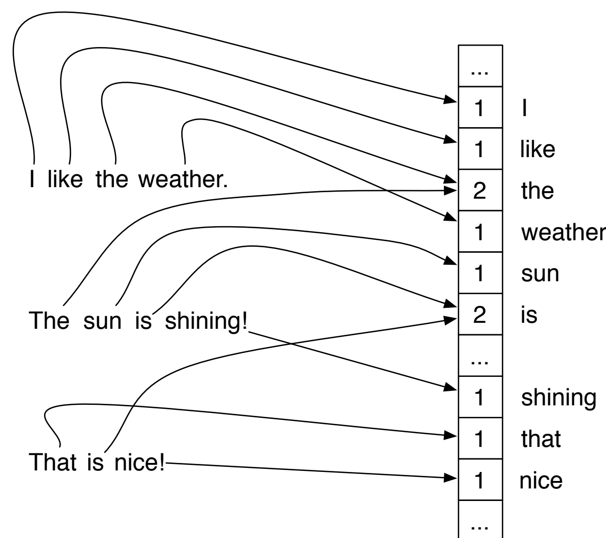

Figure 2.3: Illustration of the unigram bag-of-words vector of a text using term frequency as values.

Considering informal short texts like tweets, the data sparsity increases with higher order of n-grams. The probability for a bigram to be seen in the training phase is significantly smaller than for an unigram. Hence, bigrams or higher order n-grams are often not suitable to be used as stand-alone features. Thus, most of the time the unigram word model is used. Nevertheless, bag-of-word models with unigram features include some very naive assumptions. First of all, it is assumed that the order of the words is irrelevant, and that the words have no interconnection. Unfortunately, this model is not able to capture negations. Within the context of sentiment analysis this could be very important, because *not happy* obviously is of negative sentiment, which the simple unigram bag-of-words model is not able to capture. Still, it performs quite well for various text classification tasks, despite its naive assumptions (Joachims 2002). Moreover, it is often used as a baseline to compare with in Twitter Sentiment Analysis and plays an important role as an integral part of multiple algorithms (Go et al. 2009, Saif et al. 2012a;b, Bakliwal et al. 2012).

2.3.2 Sub-Word Level N-Grams

For sub-word level representations n-grams are also very popular (Joachims 2002). Some promising results for text classification have been reported by Neumann and Schmeier (1999). In contrast to word level n-grams the text window does not move wordwise, but characterwise. Hence, the building blocks of the model are now groups of n characters, and not n words anymore. The string *„computer"*, split up into trigrams ($n = 3$), results in the tokens: *_co, com, omp, put, ute, ter, er_*. The beginning and the end of a word are often marked with an underscore to emphasize that the n-gram did not occur within a word.

One benefit of sub-word level n-grams lies in the fact that they naturally model similar words. Take the words *„computer"* and *„computers"* as an example. Without the necessity of special linguistic analysis, the model captures the similarity of these words, because they have multiple trigrams in common. Therefore, the model would treat them very similar which obviously is the desired behavior. Furthermore, this representation is language agnostic. In some languages such as German the correct forms of words are often built in complicated ways. Using sub-word level n-grams there is no need for language specific linguistic analysis. In contrast, this behavior could also be misleading though. Words like *„computer"* and *„commuter"* also have multiple trigrams in common. In this case the effect is not desired. However, there are further benefits like robustness against spelling mistakes. Especially the informal language of tweets is full of abbreviations and spelling mistakes. Mistakes made by wrong interpretation of these similar words will probably have no significant effect once the training corpus is big enough.

Even though this model is very simple and seems to be robust against some of the major flaws of informal language, it has to the best of my knowledge not yet been used for Twitter Sentiment Analysis and its usability has to be looked at in detail.

2.3.3 Preprocessing

Many current methods for Twitter Sentiment Analysis or even text classification in general include various steps of data preprocessing. One of the most important goals of preprocessing is to enhance the quality of the data by removing noise. Another point is the reduction of the feature space's size, because some methods may struggle with large feature vectors due to limitations in computation time and available memory.

One very popular preprocessing technique is **stopword removal**. Stopwords are words which in general do not carry much meaning or sentiment, for example *the, is, at, which, on.* In the field of Twitter Sentiment Analysis those stopwords are often removed without providing any evidence that they, in fact, are useless for classification (Pak and Paroubek 2010, Bakliwal et al. 2012, Liu et al. 2012). One possible drawback of removing the stopwords could be that named entities whose names consist of such stopwords, such as *The Who* or *Take That*, could not be recognized anymore. In addition, Saif et al. (2012b) provide evidence that removal of stopwords makes classifiers perform worse. However, the reduction in corpus size is reported to be up to 38.3% (Agarwal et al.

2011). Hence, stopword removal also needs more evidence providing experiments to be able to suggest doing it.

Stemming and lemmatization are two very similar preprocessing steps, of which at least one is used in nearly all current methods of text analysis. Stemming is the process of reducing a given inflected word to its stem. The goal is to map similar words to the same stem, which is not necessarily the word's base form. For example, the words *stemmer*, *stemmed*, *stemming* would all be reduced to *stem*. The reasoning behind this reduction is that one wants to capture the sentiment of the general concept of a word and not of all its various inflections. Lemmatization on the other hand takes into account the context of the word and also performs a dictionary lookup. Whereas a stemmer would not be able to reduce the word *better* to *good*, a lemmatizer is able to do so. However, lemmatization takes much longer than stemming and may not yield any improvements. Some methods use stemming (Liu et al. 2012), others use sophisticated lemmatization (Bonev et al. 2012) and some use none of them (Saif et al. 2012a). Regardless of the fact that all the mentioned methods perform remarkably well, there is to the best of my knowledge no direct comparison of the efficiency of those provided alternatives.

Another common practice is some kind of **spelling correction**. Especially in the context of tweets, the language is informal most of the time. Hence, there are lots of spelling mistakes. Furthermore, people use lots of abbreviations due to the 140 character limit of tweets. For example they often write *thr* instead of *there*. Agarwal et al. (2011) suggest an acronym dictionary with more than 5000 expansions[2]. Unfortunately, they did not provide a statistic about the percentage of tokens which have been expanded. Moreover, twitter users tend to spell words intentionally wrong to emphasize them. One example being *loooooove*. As the number of repeated letters can be arbitrary, it makes sense to normalize them. In order to be still able to distinguish the emphasized spelling from the correct spelling, the number of repeated letters is often reduced to two (Go et al. 2009, Agarwal et al. 2011, Saif et al. 2012a;b). The misspelled word *loooooove* would become *loove* and so would *looooooooooooooooove*. Saif et al. (2012a) report the reduction of the vocabulary size to be 3.48% on their corpus, Go et al. (2009) achieved 2.77%.

Named entity replacement can also make a model more robust. Tweets often contain names of entities, like locations, people or companies. In general, one does not want the model to learn a sentiment towards a certain entity. If for example a company had very bad press in the time frame where the training data was collected, the model would always interpret it as negative. Moreover, if the sentiment towards this company shall be tracked, it would probably never change. One possible solution to this is to replace these entities with wildcards, for example the word *London* would be replaced with *||LOCATION||* (Bonev et al. 2012). Other methods ignore nouns in general since they are of the opinion that nouns do not carry any sentiment anyway (Bakliwal et al. 2012). However, there is no sound evidence regarding this hypothesis yet. Moreover, tweets contain specific entities like mentions of other users, starting with @, or URLs. It is a common practice to also replace those with wildcards, such as *||URL||, ||USERNAME||* (Go et al. 2009, Pak and Paroubek 2010, Liu et al. 2012, Saif et al. 2012a). Regarding replacement of mentions, Go et al. (2009) report a reduction in vocabulary size by

[2] compiled from `http://www.noslang.org`

43.42%, Saif et al. (2012a) report 28.58%. The removal of URLs reduced the corpus Go et al. (2009) by 9.41%, the vocabulary of Saif et al. (2012a) became 2.91% smaller. It would be interesting to investigate if the different approaches differ significantly in performance.

In conclusion one could say that many preprocessing techniques have been tried but there is no hard evidence which of them are actually useful, and how much of a difference they really make.

2.3.4 Part-of-Speech Tagging

Part-of-Speech Tagging (POS tagging), also often called grammatical tagging or word-category disambiguation, refers to the process of tagging a word in a text as a certain part of speech, depending on its original definition and its context. Most people learn to identify nouns, verbs, adverbs and adjectives at school, which are just a small subset of what current POS taggers are able to tag. Figure 2.4 shows a screenshot taken from an online POS tagger for the sentence *„Oh man, I really like this new smartphone!"*.

Figure 2.4: The sentence *„Oh man, I really like this new smartphone!"*, POS tagged by an online demo of the University of Illinois (`http://cogcomp.cs.illinois.edu/demo/pos`, also see Roth and Zelenko (1998)).

The used POS tagger is able to identify a total of 47 tags, some are shown in figure 2.4. First of all, the tag *UH* at the word *„Oh"* means interjection, a word which expresses an emotion but is also often used to fill pauses. Besides this, the tagger is able to identify different types of nouns. Looking at the words *„man"* and *„smartphone"* one notices they are both tagged with *NN*, which means singular noun. The word *„I"*, which is also a noun, is tagged with *PRP*, meaning personal pronoun. The complete list of tags is available on the project's website[3].

Even though the aforementioned online demo is intuitively accessible for humans, it lacks an application programming interface (API). Hence, another POS tagger named **TreeTagger** (Schmid 1994; 1995) gained lots of popularity within the research community as it comes as command line tool working on almost all available platforms. Furthermore, it is able to handle multiple languages and can also perform lemmatization along with the tagging process. Lemmatization is the process of transforming a word to its basic form. This is illustrated in table 2.2.

TreeTagger was used by Pak and Paroubek (2010) in the context of Twitter Sentiment Analysis. First of all, they collected a corpus of 300,000 tweets, evenly distributed across the classes *positive*, *negative*, *objective*. In order to estimate the affinity to the classes

[3] `http://cogcomp.cs.illinois.edu/demo/pos`

word	POS tag	lemma
The	DT	the
TreeTagger	NP	TreeTagger
is	VBZ	be
easy	JJ	easy
to	TO	to
use	VB	use
.	SENT	.

Table 2.2: TreeTagger result for the sentence *„The TreeTagger is easy to use.”*, taken from `http://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger/`. See `http://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger/data/Penn-Treebank-Tagset.pdf` for a full list of the tags.

positive and negative, they used emoticons as noisy labels, following the approach of Go et al. (2009). Tweets containing happy emoticons like *:-)* are considered as positive, tweets containing sad emoticons like *:-(* are considered to be negative. They collected tweets from 44 newspapers to make up the objective class. Even though those labels are noisy to some extent, they may still approximate the real distribution sufficiently.

Afterwards, they tagged the corpus using TreeTagger to be able to do a pairwise comparison of the tag distribution across classes. This is done by computing the following measure for all tags:

$$P_{i,j}^T = \frac{N_i^T - N_j^T}{N_i^T + N_j^T} \ , \tag{2.35}$$

with N_i^T denoting the number of times the tag T occurs in class i. Thus, if $P_{i,j}^T$ is positive, the corresponding tag occurred more often in class i, if it is negative, the tag was more present in class j. The absolute value $|P_{i,j}^T|$ is an indicator of how big the difference actually is. Values close to zero indicate very similar numbers of occurrences, values close to 1 represent almost exclusive occurrence in one class.

Figure 2.6 shows a bar chart of $P_{s,o}^T$ comparing subjective tweets, a mix of positive and negative ones, with objective tweets. The authors observe a strong inequality of the distribution of POS tags across the two sets, and conclude that POS tags are strong indicators for determining the affinity of a tweet to one of those sets.

Regarding nouns, it can be observed that common and proper nouns (NPS, NP, NNS) tend to occur more often in objective texts while subjective texts consist of more personal pronouns (PP, PP$). A proper noun refers to named entities such as *Apple*, *Samsung* or *New York*, while common nouns refer to classes of entities, such as *city*, *planet* or *company*. These are used when describing something objectively. Subjective tweets, in contrast, often refer to their author or another related person or entity by usage of personal pronouns like *I*, *you* or *he, she, it*.

When looking at verbs it is striking that authors of subjective tweets tend to describe themselves or address their audience by using first or second person verbs (VBP). On the

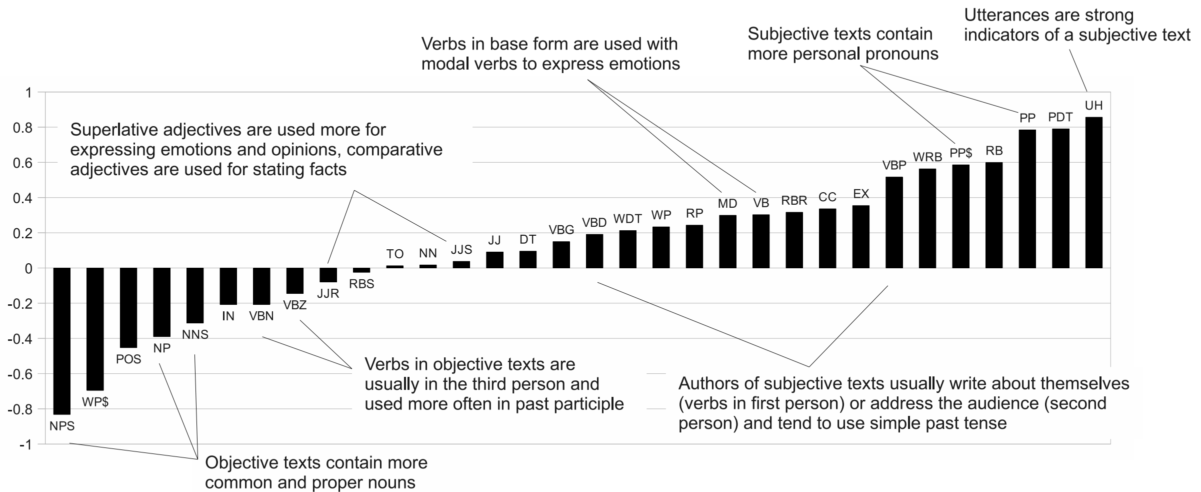

Figure 2.5: $P^T_{s,o}$ values for the classes subjective and objective, taken from Pak and Paroubek (2010).

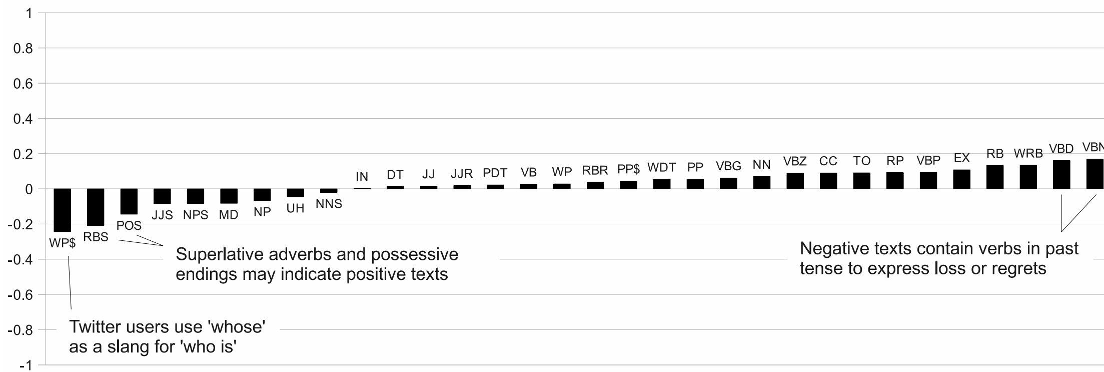

Figure 2.6: $P^T_{n,p}$ values for the classes negative and positive, taken from Pak and Paroubek (2010).

contrary, objective tweets usually include verbs in the third person (VBZ) because they are giving information about someone in general and do not directly address anybody. In terms of tenses the subjective texts more often use simple past tense (VBD) instead of past participle constructions (VBZ). In addition, the basic form of verbs (VB) is often used in subjective texts. This could be explained with the frequent use of modal verbs (MD), such as *may, might, must, shall, should*, which require another verb in infinitive form.

Even though adjectives are not distributed as clearly as nouns and verbs, it is noticeable that superlative adjectives (JJS) are used more often to express subjective emotions and opinions, whereas comparative adjectives (JJR) are harnessed to state facts or give information in an objective manner. Adverbs (RB) are found mainly in subjective texts, as their main purpose is to give an "emotional color to a verb" (Pak and Paroubek 2010).

Figure 2.6 shows $P^T_{n,p}$, the comparison between negative and positive tweets. First of all, it is remarkable that the tags are not as discriminating as for $P^T_{s,o}$. The relative differ-

ences between the occurrences in the two classes are much smaller overall. Nevertheless, there are tags like superlative adverbs (RBS), such as *most* and *best*, which significantly occur more often in positive tweets. Another very discriminating tag is POS for possessive endings. One possible explanation for this could be that people seldom tweet about things they own, but which they do not like. Tweeting about those would be the same as admitting one made a wrong decision when buying the product. In general, many people probably do not like to admit that they made a wrong decision.

In contrast, negative tweets tend to contain verbs in past tense (VBN, VBD). Pak and Paroubek (2010) suggest as a possible explanation that authors of negative tweets often express some kind of loss or regret from the past. Furthermore, they list some examples of the most frequent of those verbs, such as *missed, bored, gone, lost, stuck, taken.*

Finally, the authors mention the tag WH$ (possessive wh-pronoun *whose*) as especially interesting. The positive set of tweets has a very high occurrence frequency of the tag, which was unexpected to them. Looking at the corpus they discovered lots of tweets like the following:

dinner & jack o'lantern spectacular tonight! :)
whose ready for some pumpkins??

This reveals that people tend to use *whose* as slang for *who is*, instead of its original possessive meaning. Nevertheless, even though TreeTagger was not developed to handle informal language and slang, it seems to perform reasonably well as a generator for discriminative features for Twitter Sentiment Analysis.

To overcome the issues of TreeTagger, Saif et al. (2012b) suggest to use a tagger called **TweetNLP**, which was developed at the Carnegie Mellon University specifically for tagging tweets[4]. For details on its construction and progress see Gimpel et al. (2010), Owoputi et al. (2013). The creators of the tagger write on their website:

> We provide a fast and robust Java-based tokenizer and part-of-speech tagger for Twitter, its training data of manually labeled POS annotated tweets, a web-based annotation tool, and hierarchical word clusters from unlabeled tweets.

The tagger comes with a Java API and also brings a comfortable command line tool. It is able to handle the informal language of tweets very well. There are even special POS tags for tweet specific tokens like emoticons, hashtags, mentions and URLs.

ikr	smh	he	asked	fir	yo	last	name	so	he	can	add	u	on	fb	lolol
!	G	O	V	P	D	A	N	P	O	V	V	O	P	∧	!

Table 2.3: Example of a tweet tagged with TweetNLP, taken from Owoputi et al. (2013).

Table 2.3 shows a rather extreme example of nonstandard orthography, abbreviation and misspelling. Its meaning is basically *He asked for your last name so he can add you on Facebook.*. The tagset differs from the one of TreeTagger, which is using the Penn

[4]http://www.ark.cs.cmu.edu/TweetNLP/

Treebank-style tagset (PTB). A complete list can be found in the appendix of Owoputi et al. (2013). However, recently a PTB-style tagset was released for TweetNLP, in case one wishes to use that instead[5].

The tagging works remarkably well. For example the abbreviation *ikr*, which means *I know, right?* is correctly tagged as an interjection (!). Besides this, the phrase *yo*, meaning *yours*, is recognized as a possessive pronoun (D). As a final example, it even recognized *fb* as an abbreviation for *Facebook* and tags it as a proper noun (∧).

In conclusion, POS tagging can provide valuable features for Twitter Sentiment Analysis, which is shown by its incorporation in multiple current methods (Barbosa and Feng 2010, Pak and Paroubek 2010, Agarwal et al. 2011, Saif et al. 2012b, Bakliwal et al. 2012). Although POS tags are not used as stand-alone features, they often yield improvements when harnessed additionally to n-grams or other language models. Still, to the best of my knowledge there is no investigation comparing the performance of the sentiment classification with regard to the choice of tagger and tagset. Thus, the performance has to be tested and compared.

2.4 Overview of Current Research

In this section, an overview of current research of Twitter Sentiment Analysis is given, clarifying which combinations of classifiers, preprocessing and features have been used successfully so far. Only publications with objectives closely related to the objectives of this thesis are listed. Other approaches are not considered here since the focus of this thesis is to take a closer look at the standard approaches first. Furthermore, the datasets used to acquire those results are looked at and discussed.

One of the earliest works is Go et al. (2009). In this work, the two class classification problem, distinguishing between positive and negative sentiment, is solved. They transferred the usage of emoticons as noisy labels (see also section 3.1) from Read (2005) to the domain of Twitter Sentiment Analysis. Overall, they collected 1.6 million tweets as training data, labeled by recognition of emoticons. Regarding preprocessing, they used spell correction and replaced the tweets specific entities with wildcards. They evaluated unigram, bigram and POS tag features for SVMs, Naive Bayes and Maximum Entropy classifiers. SVMs performed best using unigrams with an accuracy of 82.2%. Using Naive Bayes classifiers with La Place smoothing and a combination of unigram and bigram features achieved an accuracy of 82.7%. The lead was taken by the Maximum Entropy classifier, reaching 83% accuracy, also using the combination of unigrams and bigrams. However, the authors do not provide any information on the significance of the differences. Furthermore, their test dataset consists of only 359 tweets. Additionally, they do not describe how exactly the test data was created or make any statement about its quality. The results are at best an indication what may perform well and sound conclusions can not be drawn from them.

Pak and Paroubek (2010) follow the approach of Go et al. (2009) using emoticons as noisy labels for training data. In addition, they are using the same dataset to evaluate

[5]See `http://www.ark.cs.cmu.edu/TweetNLP/` for more information.

their method. Hence, the credibility of the results suffers from the same drawbacks mentioned before. Regarding preprocessing, tweet specific entities along with emoticons have been removed from the training data. Furthermore, stopwords are also excluded. Basis of the method is a Naive Bayes classifier with a combination of bigrams and POS tags as features, smoothed using Jelinek-Mercer smoothing with $\alpha = 0.5$ (each set of features contributes equally to the probability for a class). Their goals were not to find an algorithm outperforming others but to investigate the usability of features. They found that bigrams perform better than unigrams and trigrams and that large numbers of training samples increase the classification accuracy. Finally, they introduced two measures to determine the quality of features, which did not lead to an improvement in general.

Agarwal et al. (2011) introduced a special set of features for SVMs, called **senti features**. Those senti features can be natural numbers, like the count of negations within the tweet, real numbers, such as the percentage of capitalized text, or binary, for example the presence of an exclamation mark. Incorporating their senti features along with the unigram features raised the accuracy by about 4%. However, no significance intervals are given here and the overall accuracy is just 75.39%. As testset they acquired 11.875 hand labeled tweets from a commercial source. Those tweets were collected by sampling the Twitter API, translating them using Google Translate[6] and finally letting people label them. After removal of tweets which have been labeled as *junk*, 8752 tweets were left to work with. However, the translation step is fairly questionable. Even though Google Translate works remarkably well, a translated tweet will differ a lot from the same tweet originally written in English. For example, Google Translate would never output the informal language most authors of tweets use. Hence, this dataset is also not suitable for a sound and general comparison of methods. However, the results suggest that handcrafted features for SVMs can possibly yield improvements.

Liu et al. (2012) introduced the idea of using a combination of noisy labels like emoticons and hand labeled data. A Naive Bayes classifier is the basis for their method. Using Jelinek-Mercer smoothing, they combine a hand labeled training set and a emoticon model with the features being unigrams only. In addition, tweet specific entities were replaced with wildcards, stopwords were removed and stemming was performed. Both classification problems, the subjectivity classification (distinguishing between neutral tweets and those carrying sentiment) and the two class classification of positive and negative sentiment (polarity classification) were investigated. While the distant supervised language model (emoticons) achieved an accuracy of only 72%, the incorporation of just 768 hand labeled tweets raised this up to about 82% for polarity classification. The results for subjectivity classification are very similar. As a dataset, they used the Sanders corpus[7], consisting of 5,513 hand labeled tweets. After filtering out non-English and spam tweets, 3,727 tweets were left for experimenting. This fact alone lets arise doubts about the quality of the dataset. How can there still be spam tweets in there, when it was hand labeled? Furthermore, those tweets were collected querying the Twitter API for only four keywords (Apple, Google, Microsoft and Twitter). Thus, it is very biased towards tweets about those entities. Nevertheless, it serves the purpose of vali-

[6] `http://translate.google.com`
[7] `http://www.sananalytics.com/lab/twitter-sentiment/`

dating they ideas of Liu et al. (2012). But for a general comparison to other methods it is not suited well.

Saif et al. (2012a) also use a Naive Bayes classifier with Jelinek-Mercer smoothing. Their basic model are unigram features, smoothed with semantic features. They found what they call **sentiment-topic features** to perform best. Sentiment-topic features are generated by a clustering method for words, which clusters them by sentiment and topic. The unigram model is then augmented with the sentiment topics present in the tweets. For preprocessing, they used tweet specific entity replacement and repeated letter spelling correction. To compare their approach with others, the dataset of Go et al. (2009) has been used. Although the approach outperforms the other approaches by a few percent (it achieves 86.3% accuracy), the test suffers from the afore mentioned drawbacks. Furthermore, no statements regarding significance of the differences are made. However, Saif et al. (2012a) have done some pioneer work regarding the integration of semantics into Twitter Sentiment Analysis. Yet, semantic features are a recently emerging approach which will not be investigated in this thesis.

Finally, the work of Bakliwal et al. (2012) is the one closest related to the work done in this thesis. The authors are using an unigram model as baseline. Afterwards, preprocessing techniques are incorporated step by step to monitor their effectiveness. Unfortunately, there are no mentions of significance tests. However, their results indicate that spelling correction, stemming, and stop word removal increase the accuracy. The best results were achieved by SVMs in conjunction with sentiment features vectors, similar to those of Agarwal et al. (2011). On the dataset from Go et al. (2009) the method reached 88% accuracy. Futhermore, another dataset called *Mejaj* (Bora 2012) is used. It is very similar to the other one, but instead of using emoticons as noisy labels, it uses a handcrafted list of 40 words, 20 for each sentiment. Those test sets are neither very large, nor is there any information about the hand labeling process and the quality criteria. Thus, even though the results are interesting indications of the authors claims to be correct, a final conclusion can not be drawn without further evidence.

Finally, one has to draw the conclusion that the evaluation methodology seems to be overall lacking. Researchers seldom provide information on how exactly the test data was labeled. Probably every tweet was only labeled by one person. Moreover, the datasets used are often very small, in the range of 300-500 tweets in total. Hence, outperforming another method by 3% just means it classified nine more tweets correctly, which is not that much of a difference. To conclude, there are lots of contradictory results regarding various hypothesis about preprocessing and feature selection. As it would go far beyond this thesis to take a closer look at all the afore mentioned approaches, only a few can be analyzed.

3 Performance Investigation

First of all, the construction of a high quality dataset is described and it is analyzed with regard to the various preprocessing techniques and features introduced in the previous chapter 2. Afterwards, the methodology to compare the various combinations of classifiers, features and preprocessing techniques will be explained. Finally, the comparison's results are presented and discussed.

3.1 Construction of a General Purpose, High Quality Dataset

At first, the quality criteria for a test dataset are discussed, followed by an explanation of the methodology for constructing such a dataset. Finally, some statistics about the dataset are provided and discussed.

3.1.1 Quality Criteria

The analysis of a dataset's quality for the general problem of Twitter Sentiment Analysis is mostly neglected in current literature (see also section 2.4). Any experimental results can only be as good as the quality of the test used to measure performance. In conclusion, before starting any experiments, quality criteria for the dataset have to be defined and respected when creating it.

Primarily, a desirable feature of a testset is to reflect human judgement. The aim of the whole field of Twitter Sentiment Analysis is to create algorithms which are able to decide about a tweet's sentiment as a human judge would. Thus, it is crucial that the testset is labeled by humans. Another approach would be to use multiple different classification algorithms and just take into account the tweets those algorithms all agree on. However, the multi-algorithm approach is not a good idea, because it is not guaranteed that such a dataset represents human judgement well.

Moreover, the size of the dataset plays an important role. Comparing the performance of two algorithms with a dataset of only 300 tweets is not very meaningful. Outperforming another algorithm by 3% means only nine more tweets have been classified correctly. The reason for not providing any confidence intervals for those experiments probably is that the results are not statistically significant due to the small size of the dataset (see also section 2.4). Hence, the dataset should be as large as possible to be able to conclude useful insights from the experiment's results.

When creating a dataset one has to keep in mind that not even two humans are able to agree on the sentiment for every tweet they are presented with. Therefore, every tweet should be rated by multiple people. Only tweets for which the various human judges reach agreement about the sentiment should make it into the dataset. For

the same reason, more than two people should label the dataset. To reach a broader representation of human sentiment judgement, the labeling should be done by as many people as possible.

Another point is topical bias. Assumed, the tweets of the dataset have been collected by querying the Twitter API for a given set of keywords, all tweets in the dataset would be about the same few topics. Hence, the actual test results would not imply the algorithm's performance of identifying sentiment of tweets, but identifying the sentiment of tweets about certain topics. While this behavior could be desired for some experiments, for general investigations regarding Twitter Sentiment Analysis it is not. But even when collecting randomly sampled tweets from the Twitter API, one has to think about the time frame. While one could collect more than enough tweets at one day, it is not advisable to do so. On one particular day, some event with great public impact, such as the Eurovision Song Contest for example, could have taken place. Thus, a large majority of tweets would be just about this topic and also result in strong topical bias. To solve this problem, one could just sample tweets over a longer time frame.

An informal experiment, which I conducted by looking at tweets from the random stream, suggested that most of these tweets are about peoples everyday life. It could be beneficial for the dataset to manually inject some tweets about certain topics like events, companies and products. So the test would reflect a broader variety of topics and would not be biased towards people's everyday life.

To conclude, the dataset should be reasonably large, labeled and validated by multiple people and the tweets used should be collected within a wide time frame, in order to make sure there is no topical bias.

3.1.2 Labeling the data

Within the course of another research project[1] about 43.5 million tweets have been collected. Of these, 33 million are randomly sampled from the Twitter API, the remaining 10.5 million tweets are focussed on certain events, such as the WWDC[2], E3[3], the CEBIT[4] or the GoogleIO[5] and others. These events also cover certain products and companies, such as Google, Samsung, Apple, Sony, the iPhone, the XBox, the Playstation and others. All those tweets have been collected between June 2012 and August 2013. Hence, when sampling randomly from this collection, the resulting dataset fulfills all the afore mentioned quality criteria.

Once the data has been set up, an interface for the labelers has to be provided to tag tweets with their sentiment, so the labeling process can be done comfortably. To achieve this, a Ruby on Rails[6] web application has been implemented. Using a web application makes the labeling process easily accessible for the labelers. Ruby on Rails is the tool

[1] `http://datamining.informatik.uni-osnabrueck.de/de/Start.html`
[2] `http://www.apple.com/de/apple-events/june-2013/`
[3] `http://www.e3expo.com`
[4] `http://www.cebit.de`
[5] `https://developers.google.com/events/io/`
[6] Ruby on Rails is a full stack web application framework, see `http://rubyonrails.org`

of choice here because it is one of the best frameworks to create stable applications in short time, see Haldenwang (2011) for a more in-depth discussion on this topic. The application has to fulfill the following requirements:

- An interface has to be provided to import the sampled tweets that should be labeled.
- The effort for the labeler has to be minimal. People tend to get inattentive when the work they are doing is to uninteresting and takes too long.
- A tweet should be labeled with one of the following labels: *positive, negative, objective, dismissed*. The label *dismissed* is used for tweets for which the labeler was not able to decide of which sentiment the tweet was. This may happen if he did not understand its meaning, or because it is just not decidable. Obvious spam tweets written by bots also have to be dismissed.
- To make full use of the labelers workforce, a tweet has to be tagged with one or more topics from the following list: *Celebrity, politics, product, company, wisdom_and_quote, event, entertainment, misc*. The category *politics* represents politicians, political events and political decisions. Sports, music, films and other entertainment related entities are represented by *entertainment*. For tweets which do not fit under any of these topics, the category *misc* has been introduced. This data can then be used to observe the topic distribution and may also be used for other experiments related to a tweet's topic.
- Once enough tweets are labeled, a second validation pass has to be performed. Each tweet needs to be labeled again by a second person other than the initial labeler. Only those tweets for which the initial labeler and the validator agreed upon the tweet's sentiment make it into the testset.
- Each labeler has to login using a user name and a password. This serves two purposes. First of all, it prevents unauthorized access to the application so that only trusted labelers can label tweets. Secondly, in order to assign a correct tweet for the validation phase, the application has to know who is labeling right now.

At first, a data model has to be defined. Figure 3.1 shows an Entity Relationship Diagram (ERD) of the resulting data model. It basically consists of two entities, *Tweet* and *Category*. The entities are connected with a many-to-many relationship because a tweet may be about more than one topic. The attribute *content* includes the tweets text. In *original_id* the tweets *id* generated by twitter is saved. This *id* is needed to make sure there are no duplicate tweets in the dataset. A simple unique index ensures this automatically while importing the data. The attribute *created_at* represents the time the tweet was written. Within the field *label* the label assigned in the first phase is saved, whereas *label_validation* represents the label given by the second labeler in phase two. Accordingly, *labeler* holds the user name of the initial labeler, whereas *validation_labeler* includes the validators user name. One could have introduced another entity such as *Label* and create a one-to-many relationship with *Tweet*. But for reasons of simplicity this has been consciously denormalized.

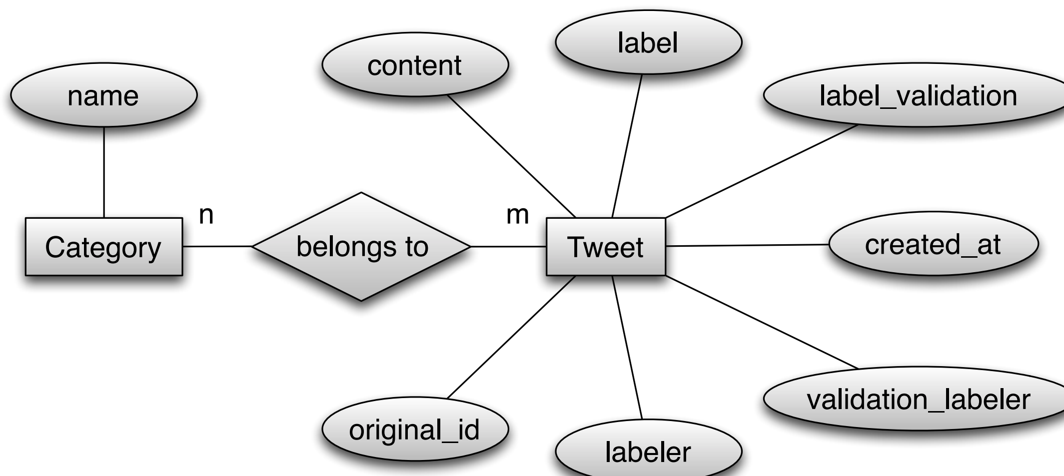

Figure 3.1: Entity Relationship Diagram for the labeling applications data model. Primary keys are artificial *id* attributes. They are not shown here.

Regarding the import, one has to provide a JSON[7] file in the format shown in listing 2, which is basically an array of JSON objects containing the fields *content* and *original_id*. This file has to be placed within the applications `data` directory. Since this step is only performed once, there is no need for a web interface. The simple command line invocation `rake import` starts the import process. Tweets already existing in the database, identified by their *original_id*, are not imported again.

```
1 [
2   {
3     "content": "She cheated the English test :P",
4     "original_id": 229894936642859008
5   }
6 ]
```

Listing 2: Example of format for a JSON data file that can be imported, containing one tweet.

Finally, the user interface has to be developed. When visiting the applications URL, the user has to authenticate with basic HTTP authentication[8]. Afterwards he is presented with the labeling interface, which is the only page of the application. Figure 3.2 shows a screenshot of the designed interface. At the top of the page, the user is shown the tweet to be labeled. Directly below that, there are four buttons indicating the possible labels with intuitive colors. Green is for *positive*, blue is for *objective*, red is for *negative* and black is for *dismiss*.

Before pressing one of those buttons to get to the next tweet to be labeled, the user has to pick the categories. As default, *misc* is selected, as this fits the majority of

[7] `http://www.json.org`
[8] `http://en.wikipedia.org/wiki/Basic_access_authentication`

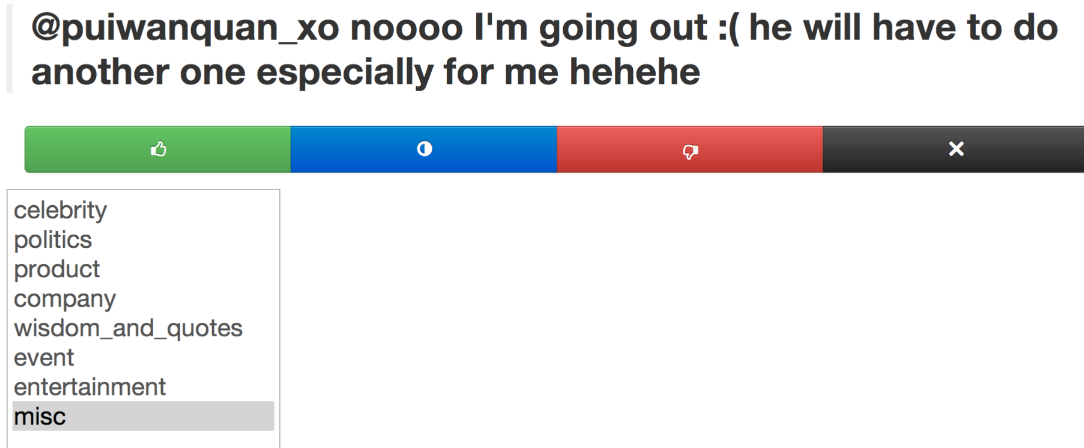

Figure 3.2: Screenshot of the labeling web interface.

tweets. Hence, in most cases, the user does not have to pick any category. However, when the category is not *misc*, another one can be selected by just clicking on it. To select multiple categories, one has to hold down the control key (CTRL) while clicking. Finally, when the user has finished, he presses one of the buttons and is automatically presented with the next tweet. The interface stays exactly the same for both phases. The idea behind that is that the validator should not know which label the tweet initially has been labeled with in order to minimize any bias. To switch the mode an internal flag within the application has to be changed. This influences the choice of the presented tweets according to the above mentioned requirements and saves the label to the corresponding attributes.

When both phases, initial labeling and validation, are completed, the data can be exported via another command line invocation: `rake backup:create`. The format is very similar to that shown in listing 2, but additionally includes the attributes presented in figure 3.1.

3.1.3 Test Dataset Statistics

Overall, 10,176 tweets have been labeled by 23 labelers, including one researcher from the Institute of Computer Science of the University of Osnabrück, 21 students who took part voluntarily and myself. The results of the initial labeling are presented in table 3.1. From those 10,176 labeled tweets, 5,726 (56.2%) have been discarded. This large amount of unusable tweets highlights once again how tedious it can be to acquire a dataset of high quality. Less than half of the tweets looked at can possibly be kept.

It is interesting that the majority of the non-dismissed tweets (38.4%) is of negative sentiment. One possible explanation for this could be that people tend to complain about negative experiences rather than writing about positive ones. For example, nobody would tweet that he received his parcel in time, whereas people tend to complain when it is delayed. Another surprising fact is that positive tweets (29.1%) are even outnumbered by the objective ones (32.4%).

Label	#	% non-dimissed
positive	1,296	29.1%
negative	1,711	38.4%
objective	1,443	32.4%
dismissed	5,726	-

Table 3.1: Distribution across labels after initial labeling. The last column shows the percentage of the label in relation to the number of non-dismissed tweets.

Label	#	% validated	% retained
positive	910	28.4%	70.2%
negative	1,328	41.4%	77.6%
objective	966	30.14%	66.9%

Table 3.2: Label distribution across labels after validation phase. The second column shows the labels percentage of successfully validated tweets, the third column shows the percentage of tweets that have been retained from the original labeling.

Table 3.2 shows the class distribution after the validation phase. Of the 4,450 initially non-dismissed tweets, 3,204 (72%) could be retained, while 1,246 (28%) had to be discarded due to validation labeler disagreement. The distribution across classes shifted slightly more towards the negative class, which makes up 41.4% (38.4% after initial labeling) of the non-discarded tweets. Hence, the other two classes slightly decreased in size with positive class still presenting the minority. The negative class has the lowest disagreement. Overall, the validators agreed with the initial labelers on 77.6% of the negative tweets. For the positive class this were just 70.2%, objective tweets were agreed on the least with just 66.9%. This indicates two interesting conclusions. Firstly, two humans were able to agree on a tweet's sentiment only for 72% of tweets. Hence, an algorithm achieving an accuracy above 72% can be considered performing reasonably well. Secondly, the classification of a tweet as negative seems to be easier for humans and is more commonly agreed upon. Thus, an interesting question is: Will this be the same for algorithms?

Initial Label	↯ Positive	↯ Negative	↯ Objective	↯ Dismissed
positive	-	78 (6.0 %)	279 (21.5 %)	29 (2.2 %)
negative	89 (5.2 %)	-	267 (15.6 %)	27 (1.6 %)
objective	195 (13.5 %)	220 (15.2 %)	-	62 (4.3 %)

Table 3.3: Disagreement matrix. Rows denote the initial label, columns the number of conflicted validations with the respective validation label.

Due to the surprisingly high rate of disagreement, a closer look at the conflicts promises interesting insights. Table 3.3 presents a disagreement matrix. The rows represent the initial label, the columns the validation label. The elements in the matrix indicate the number (and percentage) of tweets which were initially labeled with the row label, and were labeled with the column label in the validation phase. This data also reveals interesting facts. First of all, the rate on which initially labeled tweets are dismissed is fairly small (2.2%, 1.6% and 4.3%). Thus, all labelers seem to have a similar understanding of which tweet's sentiment is undecidable or which tweets are spam. Moreover, conflicts between negative and positive are also fairly small. Only 6.0% of the validated tweets have been initially labeled positive and got negative as validation class. The other way around, only 5.2% were initially labeled negative, and have been validated as positive. Thus, humans mostly seem to agree upon those two, very opposite classes.

However, distinguishing those from objective tweets seems to be much harder for humans. The disagreement rate between positive/negative and objective extends from 13.5% to 21.5%. This kind of raises the bar for algorithms performing two class classification between negative and positive tweets, as two humans agree upon such a tweet's sentiment in about 89% of the time.

Topic	Before Validation	After Validation	Retained
misc	2,565	1,896	73.9%
product	659	489	74.2%
event	421	318	75.5%
company	324	231	71.3%
wisdom_and_quotes	302	183	60.6%
celebrity	284	194	68.3%
entertainment	159	102	64.2%
politics	92	64	69.6%

Table 3.4: Topic distribution before and after validation with percentage retained after validation.

Finally, some statistics are presented on the distribution across topics in table 3.4. The first column denotes the topic's name, in the second column the number of tweets tagged with the topic in the initial labeling phase is given, column three shows the numbers of non-conflicted tweets after validation and the final column presents the percentage of tweets retained after the validation phase. Note that a tweet can be tagged with multiple topics, hence, the numbers do not sum up to the numbers of tweets in the testset. As expected, the topic *misc* is prevalent. It is neither very surprising that *politics* is the topic with the least taggings. Twitter is mainly used by relatively young people who are, according to the general opinion, not very interested in politics. Hence, sampling randomly from the twitter stream does not yield many tweets about politics. However, it is interesting that *product* is the second place. This is an indicator that harvesting tweets for marketing purposes can be very fruitful. Furthermore, tweets about a product have a fairly high rate of sentiment agreement (74.2%). Topics like *company* or *event* are not as present but their agreement rate is still decent. However, deciding on the sentiment of tweets about topics such as *wisdom_and_quotes, celebrity, entertainment* and *politics* seems to be harder, the agreement rate is below 70%. Although the topic data is not used in this work, it still provides some interesting insights and may be useful for other investigations in the future.

It would also be interesting to perform an analysis of how well the features introduced in section 2.3 separate the classes from one another. Unfortunately, the number of tweets is still very small. Some informal experimentation revealed that most of the tokens occur only once and hence the results of such an investigation would not provide further insight. However, the following section presents such an analysis on a fairly large training corpus.

3.1.4 Collection and Analysis of Training Data

Collecting training data is done using the approach of Go et al. (2009). The idea is to use emoticons as noisy labels. Tweets including happy emoticons like *:-)* have a high probability to be of positive sentiment, whereas tweets including sad emoticons like *:-(* tend to be negative. As discussed in section 2.4, this method has already been validated multiple times, by using this training data to evaluate hand labeled test data. One million positive and one million negative tweets were extracted from the 43.5 million tweets corpus introduced in section 3.1.2, using the emoticon list shown in table 3.5.

<table>
<tr><th>Positive Emoticons</th><th>Negative Emoticons</th></tr>
<tr><td>:], :-), :), :o), :], :3, :c), :>, =], 8), =), :}, :^), >:D, :-D, :D, 8-D, 8D, x-D, xD, X-D, XD, =-D, =D, =-3, =3, 8-), :-)), :*, >;], ;-), ;), *-), *), ;-], ;], ;D, ;^), >:P, :-P, :P, X-P, x-p, xp, XP, :-p, :p, =p, :-b, :b, :'-), :')</td><td>:[, :-(, :(, :-c, :c, :-<, :<, :-[, :[, :{, :-||, :@, D:<, >:\\, >:/, :-/, :/, :\\, =/, =\\, :S, :'-(, :'(</td></tr>
</table>

Table 3.5: Emoticons used as noisy labels, separated by commas.

Acquisition of objective training data poses a yet unsolved problem. One common approach is to consider tweets of accounts from newspapers as objective. However, those tweets often contain the headlines of news articles which carry certain sentiments, depending on the news. For example, the headline *There were 43 innocent children murdered for no reason!* obviously would be considered negative by the majority of humans. Hence, this work is concentrated towards discriminating only between positive and negative tweets because no objective training data could be obtained.

$\mathcal{X}^2$ Analysis of Feature Types

First of all, a so called $\mathcal{X}^2$ **analysis**[9] is performed for each feature introduced in section 2.3. The test provides insight into whether the difference in occurrence of a feature for the two classes positive and negative is statistically significant. Features with occurrence frequencies below five can not be handled and hence are discarded for the test. For example, if a word occurs twice in a positive context and once in a negative context, this value would not be considered significant. In contrast, if the word occurred 2,000 times in a positive context and 1,000 times in a negative one, the difference would be significant. As confidence interval, the standard $\alpha = 0.05$ is used, meaning a difference is considered significant if its probability of being random is below 5%.

Results of the analysis and some general statistics per feature are presented in table 3.6. The first column denotes the name of the feature, in the second column the total number of different features is presented. The column *"Once"* indicates the percentage of features which occurred only once in the whole corpus. Column *"< 5"* presents the percentage of features which occurred more than once but less than five times and

[9] This commonly statistical test is not explained here, for details see Greenwood and Nikulin (1996).

hence also can not be considered. The column *"Insignificant"* denotes the percentage of features which occurred more than five times but with a difference in occurrences that does not significantly differ between the negative and positive classes. Finally, the columns *"Positive"* and *"Negative"* present the percentage of features which significantly more often appear in tweets of the respective class.

Feature Type	Total	Once	< 5	Insignificant	Positive	Negative
unigrams	1,560,680	78.4%	16.5%	3.6%	1.0%	0.5%
bigrams	6,772,784	78.9%	14.9%	4.4%	1.0%	0.8%
trigrams	15,751,229	84.6%	12.0%	2.3%	0.6%	0.6%
subgrams3	66,793	24.4%	19.8%	36.0%	17.2%	2.7%
subgrams4	528,192	32.3%	30.1%	29.0%	7.2%	1.4%
pos-default	26	3.8%	3.8%	3.8%	53.8%	34.6%
pos-treebank	44	0.0%	2.3%	4.5%	61.4%	31.8%

Table 3.6: $\mathcal{X}^2$ analysis of the training corpus.

This data suggests some interesting conclusions. First of all, it is very surprising that for all the n-gram feature types still so many tokens appear just once in the whole corpus. For $n \geq 2$ it was expected that the data would become more sparse. However, 78.4% of unigrams occurring just once is also very surprising. One possible explanation for this could be that named entities appear as single unigram. This includes mentions, location, product and company names, URLs and hashtags. Another possible explanation can be a vast amount of informal language, spelling mistakes, and abbreviations. For higher order n-grams this effect becomes even stronger, as those are combinations of the unigrams. Going from unigrams to bigrams, the one time occurrence rate is stable, while rising to about 6% when using trigrams. The percentage of tokens occurring less than 5 times is also above 12% for all n-gram features. Considering the average rate of insignificant features for n-grams is about 3%, there is not much room left for significant features, their rates being 1% and lower. These low rates of significant features are very surprising, as n-grams have been reported to perform reasonably well (see section 2.4).

Looking at the sub-word level n-grams (**subgrams**), one notices a much smaller percentage of insignificant features. For both, *subgrams3* and *subgrams4* the summed percentage for *Once* and < 5 is smaller than *Once* alone for the n-grams. The overall number also is reasonably smaller. Only 66,793 *subgrams3* tokens have been extracted, which is a considerable large difference to 1,560,680 unigrams (just 4% of the size). Since the rates of significant tokens is overall larger than for n-grams, one may assume that subgrams are better features for classification so far.

Finally, there are the two POS tagsets to look at. The feature type *pos-default* denotes the tweet specific tagset from TweetNLP, whereas *pos-treebank* uses a standard tagset. The number of tokens for the POS tagsets equals the number of different POS tags which appeared in the corpus. It is remarkable that most of the POS tags are significantly discriminating between positive and negative tweets. Thus, POS tags seem to be very good features for positive/negative classification.

One observation that can be made for all features is the higher rate of significantly positive tokens compared to the rate of the negative ones.

Preprocessing Statistics

Another $\mathcal{X}^2$ analysis is performed after applying the preprocessing steps introduced in section 2.3.3 to unigram features one at a time. The results are presented in table 3.7.

Acronym expansion was done with the same dictionary used by Agarwal et al. (2011), consisting of more than 5,000 slang acronyms. However, it had no statistically significant effect. Only 395 unigrams were replaced, making up just 0.03% of the vocabulary.

Similar effects are observable for stopword removal. The stopword list provided by MySQL for their full-text search engine[10] was used here. Applying stopword removal reusulted in a reduction of the vocabulary of less than 0.5%.

Spelling correction, as described in 2.3.3, achieved a reduction of 1.6%. Nevertheless, the percentage of significant tokens did not change notably.

Stemming, performed using the commonly known **Porter Stemmer** algorithm, reduced the size of about 3.8%. Unfortunately, the percentage of significantly discriminating tokens dropped further from its already low percentage.

Preprocessing	Total (Reduction)	Once	< 5	Insignificant	Positive	Negative
nothing	1,560,680 (100.00%)	78.4%	16.5%	3.6%	1.0%	0.5%
acronyms	1,560,285 (99.97%)	78.4%	16.5%	3.6%	1.0%	0.5%
stopwords	1,560,193 (99.96%)	78.4%	16.5%	3.6%	1.0%	0.5%
spelling	1,536,341 (98.4%)	78.3%	16.6%	3.7%	1.0%	0.5%
stemming	1,502,417 (96.2%)	79.1%	16.3%	3.2%	0.9%	0.4%
entities	455,309 (29.17%)	65.5%	20.6%	9.9%	2.5%	1.5%
lemmas	43,520 (2.7%)	25.1%	25.6%	32.4%	11.2%	5.8%

Table 3.7: $\mathcal{X}^2$ analysis of unigrams combined with preprocessing.

Entity replacement and lemmatization seem to be the only promising preprocessing techniques here. Replacing entities reduces the vocabulary size to 29.7% of its initial value. Moreover, the rate of tokens occurring just once is reduced by more than 10% and the percentage of significantly discriminating features also increased noticeable. The reduction of lemmatization to just 2.7% of the corpus' original size is surprising though. The lemmatizer used is the `NSLinguisticTagger`[11] provided by the Mac OS standard library. It seems to ignore words it is not able to lemmatize, hence the high reduction. However, the rates of significant tokens seems very promising.

[10] `http://dev.mysql.com/doc/refman/5.5/en/fulltext-stopwords.html`

[11] `https://developer.apple.com/library/ios/documentation/cocoa/reference/NSLinguisticTagger_Class/Reference/Reference.html`

Conclusions

To summarize the analysis of the noisy labeled training data, one could say that POS tags seem to be the features discriminating best, followed by subgrams. Regarding preprocessing, most of the techniques only affect small parts of the corpus and hence are not very promising. Nevertheless, entity replacement and lemmatization seem to have a positive impact on the classification performance. These results only reflect the dicriminativity between the two classes which have been noisy labeled with emoticons. How valuable those features really are has to be investigated by evaluating the hand labeled test data.

3.2 Measuring and Comparing Performance of Classifiers

As discussed in section 2.4, experimental results regarding Twitter Sentiment Analysis are often not directly comparable. Most of the time, this is the case because researchers do not provide any data on significance, but only report accuracy. Accuracy is the simplest measure coming to mind when measuring classification performance. It can be computed with the following formula:

$$Acc(c, w) = \frac{c}{c + w} \ . \tag{3.1}$$

The variable c represents the number of correct classifications and w denotes the number of incorrect classifications. Accuracy is not a bad measure in general and sums up the overall performance of an algorithm. However, to actually get an idea whether two results differ significantly from one another, one has to perform a statistical test.

Since two class classification can be considered as a bivariate frequency distribution with one degree of freedom (the classifier is fixed and there are two attributes for it), one can use the commonly known $\mathcal{X}^2$ **significance test**, for more details see Greenwood and Nikulin (1996). To compute the $\mathcal{X}^2$ value, one can use the following formula

$$\mathcal{X}^2(c_1, w_1, c_2, w_2) = \frac{n \cdot (c_1 \cdot w_2 - c_2 \cdot w_1)^2}{(c_1 + c_2) \cdot (w_1 + w_2) \cdot (c_1 + w_1) \cdot (c_2 + w_2)} \ , \tag{3.2}$$

where c_i denotes the number of correct classifications from classifier i, and w_i denotes its wrong classification count. The total number of observations is given by $n = c_1 + w_1 + c_2 + w_2$. Finally, to determine whether the difference is significant, one has to to compare the resulting value of $\mathcal{X}^2$ against a given threshold. For the common $\alpha = 0.05$ this is 3.84. If the resulting value is higher than this threshold, the probability of the difference in performance being random is below 5% and can be considered significant.

To get a deeper insight into an algorithm's performance, one can take a closer look at first and second order errors per class. Looking, for example, at the positive class, a classification result is called **true positive** (t_p) if the tweet has been classified positive and actually is positive. It is called **false positive** (f_p) when it is classified positive, but is negative in reality. **True negative** (t_n) denotes a tweet classified negative which

actually is negative and **false negative** (f_n) is a tweet classified negative which is actually labeled positive. For the negative class, the actual numbers are the same, but the values are mirrored. For example, the true positives of the class positive are the true negatives of the class negative. Table 3.8 provides an example.

	Positive	Negative
t_p	705	1,067
f_p	261	205
f_n	205	261
t_n	1,067	705

Table 3.8: Illustration of the per class first and second order errors.

Due to the symmetry one can compute the accuracy from this values by arbitrarily choosing one of the classes and using the following formula:

$$Acc(t_p, f_p, f_n, t_n) = \frac{t_p + t_n}{t_p + t_n + f_p + f_n} \,. \tag{3.3}$$

Using this formula, the accuracy for the example in table 3.8 would be 79.2%. However, for some applications it may be more important to minimize or maximize certain errors. For example, when a company is interested in acquiring negative feedback regarding their products, in order to be able to react to it, they do not want to miss out on any negative tweet. Hence, for the negative class the number of false positives should be small. To characterize such traits of an algorithm, the measures **precision** and **recall** have been introduced. They can be computed as following:

$$Prec(t_p, f_p) = \frac{t_p}{t_p + f_p} \tag{3.4}$$

$$Rec(t_p, f_n) = \frac{t_p}{t_p + f_n} \tag{3.5}$$

Precision is a measure for how many of the positive classifications are actually positive. To rephrase this one could say: The higher the precision, the lesser the rate of examples falsely assigned to the class. To continue the example from above, the precision for the negative sentiment class is probably not that important for the company, as they are only interested in capturing all negative tweets. Still, they are interested in a high recall. High recall means that very few tweets of the class currently looked at have been tagged with the other class. In other words: Recall indicates how many of the desired tweets are actually retrieved.

Computing the values for precision and recall for the example data in table 3.8, the positive class achieves 73.0% precision and 77.5% recall, while the negative class reaches 83.9% precision and 80.3% recall. In this case it is quite obvious for which class the classifier performs best. But, for cases where this is not obvious, the so called F_β **measure** has been introduced. It can be computed as following:

$$F_\beta(precision, recall) = (1 + \beta^2) \cdot \frac{precision \cdot recall}{\beta^2 \cdot precision + recall} \tag{3.6}$$

The parameter β can be used to put more emphasize on precision or recall. Most of the time it is set to one, hence the F_1 score is just the harmonic mean of precision and recall. For the given example this would result in an F_1 score of 75.1% for the positive class and 82.5% for the negative class.

However, in this work the per class precision and recall values are only reported and not analyzed further. The main measure will still be accuracy, in order to be able to compare the results with current methods.

3.3 Determining Training Corpus Size per Feature and Classifier

Algorithms for Twitter Sentiment Analysis have very few parameters in general. First of all, one has to choose which features to use. As soon as that is clear, a classifier has to be chosen and the needed amount of training data has to be determined. This section presents an investigation of various combinations of those parameters and it concludes with a first baseline accuracy.

The two classifiers to be looked at are SVM and NBC, both introduced in chapter 2. For each of those classifiers all features introduced in section 2.3 are evaluated with various corpus sizes. The only n-grams evaluated are unigrams, bigrams and trigrams. Higher orders of n-grams are too sparse to yield promising results. Subgrams are evaluated for $n = 3$ and $n = 4$. Informal experimentation revealed that $n = 2$ does not provide good discrimination and $n > 4$ approximates unigrams, as most of the words used in tweets are rather short. POS tags are evaluated using TweetNLP with both its available tag sets (tweet specific default and Treebank). To sum up, there are seven features to be evaluated for two classifiers, resulting in 14 experiments.

The experiment itself starts by training the classifier with a small training size, using tweets from the noisily labeled training tweets. In the next step the classifier is evaluated at the hand labeled dataset. Accuracy and per class precision and recall are reported as results. Those steps are performed for various training sizes. The first training size is 1,000. It then is increased in steps of 1,000 until 10,000. Afterwards, the stepsize is 10,000 until up to 100,000. Finally, the sizes 200,000, 300,000 and 500,000 are evaluated. The reasoning behind the non-linear increase of step size is that for smaller sizes stronger differences will occur. For each of the 14 different configurations, 23 test runs have to be performed, resulting in 322 runs in total. The significance of differences in results is computed with the $\mathcal{X}^2$ test.

Regarding SVM, only the linear kernel is used. As mentioned before, most text data is linearly separable. Moreover, the non-linear kernels take much more computation time. Where a linear kernel takes seconds to minutes for training (using LIBLINEAR), the non-linear kernels (using LIBSVM) take several hours. Hence, using non-linear kernels for this investigation would take weeks.

The NBC classifier only uses simple Laplace smoothing to deal with unknown features. At this stage of the evaluation, no fallback models are used.

To qualitatively evaluate the effects of training corpus size on the accuracy, the results of evaluation are plotted. However, not all of the 14 plots are presented here for reasons of space, just representative and particularly interesting ones are shown and discussed. Table 3.9 presents all results though. Figure 3.3 shows the plot for NBC with unigram features.

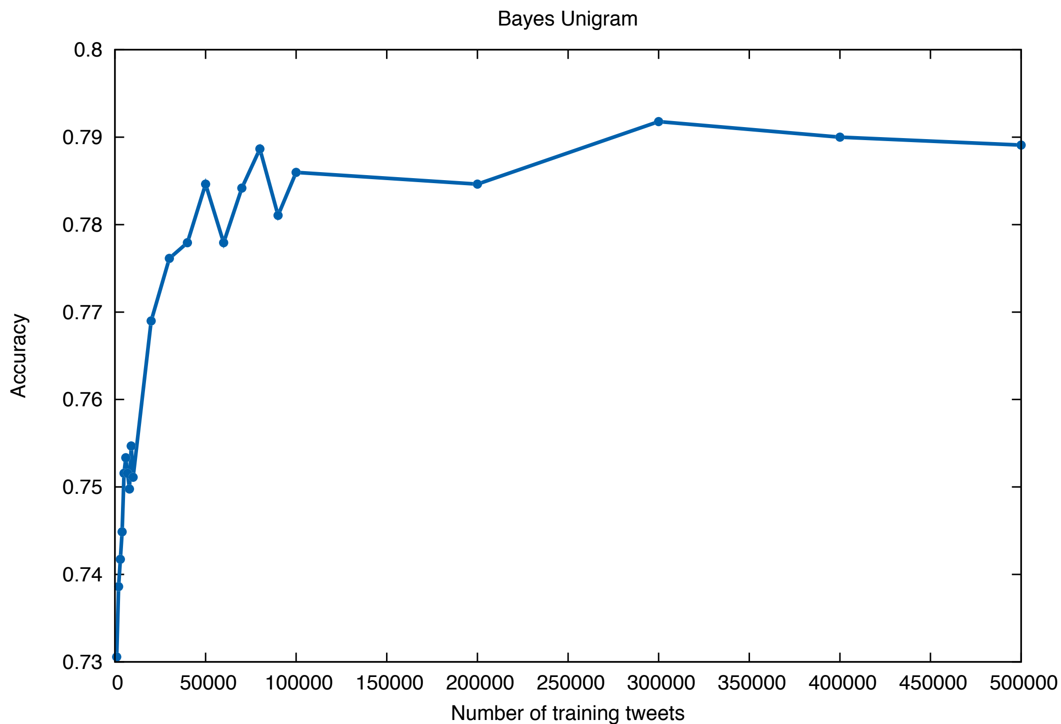

Figure 3.3: Accuracy of NBC using unigram features with increasing training corpus size.

These results reflect what was expected: The larger the training corpus, the better the accuracy. However, the improvements are no longer significant after reaching a certain threshold. A notable fact is that with just 1,000 training tweets, the classifier already achieves an accuracy of about 73%. Considering the fact that the $\mathcal{X}^2$ analysis of features revealed only about 1% of unigrams to be significantly differently distributed between classes, this is very remarkable. The first noticeable peak is reached with a training size of 50,000, resulting in 78.5% accuracy. After that the accuracy fluctuates, still not significantly. The maximum of the experiment was reached with 300,000 training tweets, providing 79,2% accuracy. Unfortunately, the differences to 50,000 and the other peak at 80,000 are not statistically significant. Differences to larger training sizes are also not significant. In conclusion, one could say that for NBC at least 50,000 tweets should be used as training data for unigram features. This is a surprising result. One would have expected that much more has to be used due to the noise within the training data. The plot for SVM unigrams looks very similar and for this reason is not presented here. Moreover, subgrams (for both, SVM and NBC) also tend to behave

quite similar, fluctuating at the beginning and then approaching a plateau, which is why the respective graphs are also not shown here.

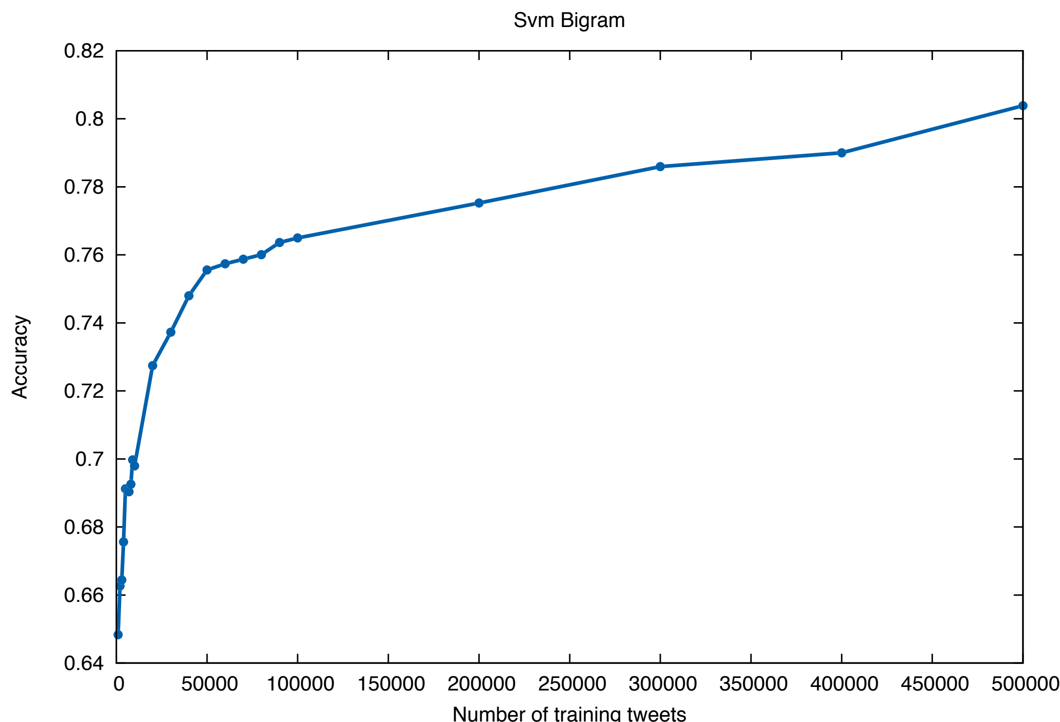

Figure 3.4: Accuracy of SVM using bigram features with increasing training corpus size.

The graph for bigram features is also fairly interesting to look at. For the SVM classifier, it is presented in figure 3.4. In comparison to the unigram graph the fluctuation is much lower. This suggests that the amount of noise is smaller. One hypothesis regarding bigrams is that they are able to capture negations. Consider for example the term *not good*. Using bigram features this would correctly be captured as negative. However, unigrams will capture a negative occurrence for *not* and a negative occurrence for *good*. Because of *good* being a genuinely positive word that is counted as negative, ignoring the negation just generates undesired noise. Using bigrams seems to prevent such noise to some extent. The accuracy almost always improves with increasing number of training tweets up to a significant maximum of 80.4% using 500,000 training tweets. The increases from 200,000, 300,000 and 400,000 are not significant statistically, whereas the jump from 100,000 to 500,000 is. This further strengthens the hypothesis of bigrams being much more sparse than unigrams because more training data seems to continuously increase accuracy, probably due to increased coverage of all existing bigrams. Even though the results suggest that further increase of training corpus size may be beneficial, this cannot be evaluated in this work due to limited computation time and memory. The graph is also representative for SVM trigrams and NBC bigrams/trigrams. Because these very similar graphs provide no further insights,

they are not presented here.

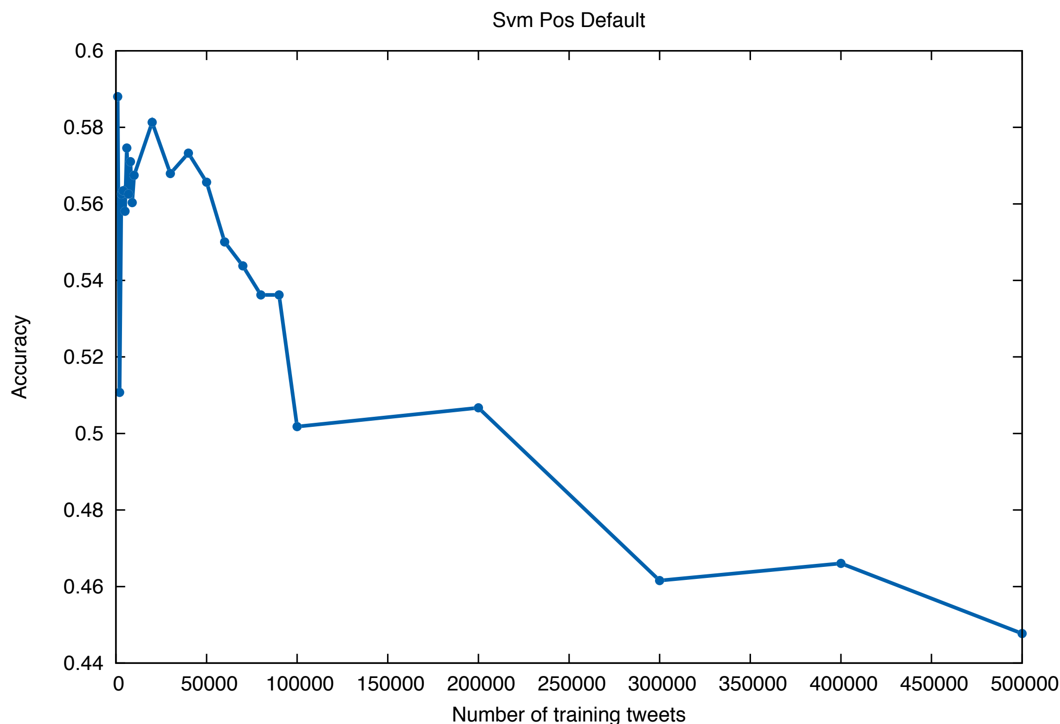

Figure 3.5: Accuracy of SVM using tweet specific POS features with increasing training corpus size.

The results when using POS tag features were the most surprising. Figure 3.5 presents the graph for SVM with POS tags using the tweet specific test set. The other POS tag related graphs look very similar and hence are left out. Right from the beginning the accuracy is rather mediocre, about 58%. With increasing number of training tweets it significantly drops below 50%. Hence, using POS tag features alone performs worse than guessing when using too many training tweets. One indication for this behavior is that POS tags have never been suggested as standalone features in literature before. Considering the results of the $\mathcal{X}^2$ analysis (see table 3.6), this is very surprising. While most of the POS tags appear significantly more often in one of the classes, using them as standalone features performs only slightly better than guessing, or even worse with too much training data. Thus, POS tags should not be used as standalone features and when combining them with other features one has to be careful not to use too much training data, since that seems to only add noise and does not seem to be beneficial at all.

Table 3.9 sums up the results of the experiment for all combinations of features and classifiers. The bold faced accuracies mark the best results. Nevertheless, NBC unigrams/bigrams, SVM unigrams/bigrams and SVM subgrams4 are statistically indistinguishable and have to be considered performing equally well. Still, looking at the

absolute values, SVM slightly outperforms NBC. Moreover, bigram features achieve higher accuracy than unigram features. This is surprising, as in literature NBC with unigrams is used as baseline most of the time. It also is stated that SVM performs significantly worse consistently. Obviously, this is not the case here, because both perform equally well. Even more interesting is the performance of subgrams4 since it is on par with unigrams and bigrams. This kind of feature has, to the best of my knowledge, not been considered for Twitter Sentiment Analysis at all. The feature space for subgrams4 is just about 30% of the size of that of unigrams, and only about 10% the size of a bigram feature space (see table 3.6). Nevertheless, it performs on par with these.

Feature	Classifier	Corpus Size	Accuracy	Prec+	Rec+	Prec-	Rec-
Unigrams	NBC	50,000	**78.5%**	71.9%	77.1%	83.5%	79.4%
	SVM	80,000	**80.0%**	72.4%	82.3%	86.6%	78.5%
Bigrams	NBC	500,000	**79.4%**	74.1%	75.7%	83.1%	81.9%
	SVM	500,000	**80.4**%	72.8%	82.5%	86.8%	78.9%
Trigrams	NBC	500,000	77.6%	71.6%	74.5%	82.0%	79.7%
	SVM	200,000	74.7%	67.4%	73.0%	80.4%	75.8%
Subgrams3	NBC	80,000	65.0%	54.1%	92.2%	89.7%	46.4%
	SVM	50,000	76.4%	69.5%	74.8%	81.8%	77.5%
Subgrams4	NBC	400,000	74.9%	63.6%	89.3%	89.9%	65.0%
	SVM	100,000	**79.0%**	73.6%	75.5%	82.9%	81.5%
POS Default	NBC	1,000	55.4%	17.5%	2.6%	57.8%	91.5%
	SVM	20,000	58.1%	48.6%	51.6%	65.4%	62.6%
POS Treebank	NBC	2,000	54.1%	46.5%	86.5%	77.5%	31.9%
	SVM	1,000	64.2%	55.6%	59.9%	71.0%	67.2%

Table 3.9: Performance for smallest training size with significant differences. The statistically indistinguishable best results are printed bold faced, the underlined values are the maximum of the respective column.

Looking at the values of precision and recall per class also reveals some interesting results. For example, NBC subgrams3 performs significantly worse with just about 65% accuracy. However, it achieves a positive recall of 92.2% and a negative precision of 89.7%. Thus, it can identify positive tweets as positive very well and does not falsely classify them as negative. However, it lacks the ability to identify negative tweets correctly. A positive precision of only 54.1% and a negative recall of 46.4% indicates that many negative tweets are classified as positive. Combining this classifier with another one that achieves high positive precision and high negative recall may yield an overall improvement.

To sum up the results, unigrams and bigrams for both classifiers and SVM using subgrams4 perform all on par. These results are statistically indistinguishable. POS tags perform worse than guessing when using to much training data. NBC with subgrams3 performs mediocre but has strong positive recall and negative precision. As a conclusion one could say that, even if canonical features are used, the classifiers performed better than expected and the values of precision and recall provide hope for overall improvement by combining classifiers according to their strengths and weaknesses. To establish a single baseline to compare to, one could compute the average of the sta-

tistically indistinguishable best results, leading to an accuracy of 79.46% (1,778 tweet classified correct, 460 classified wrong).

3.4 Effects of Preprocessing

Investigating the effects of preprocessing techniques introduced in section 2.3.3 is done by training SVM and NBC using unigram features, applying the preprocessing technique currently looked at. As size of the training corpus, the preliminary experimentally obtained minimum training size is used. Results are reported in form of accuracy and per class precision and recall. Significance of improvement is tested against the established baseline from section 3.3. Table 3.10 presents the results of the investigation.

Preprocessing	Classifier	Accuracy	Prec+	Rec+	Prec-	Rec-
Stemming	NBC	78.2%	71.4%	77.3%	83.5%	78.8%
	SVM	79.9%	72.7%	81.2%	86.0%	79.1%
Lemmatization	NBC	78.2%	73.5%	72.4%	81.3%	82.1%
	SVM	78.9%	70.9%	81.4%	85.8%	77.1%
SpellingCorrection	NBC	78.5%	72.1%	76.9%	83.4%	79.6%
	SVM	**80.5%**	73.2%	82.1%	86.6%	79.4%
NamedEntityReplacement	NBC	77.9%	70.7%	78.1%	83.8%	77.8%
	SVM	79.1%	71.9%	79.7%	85.0%	78.7%
NamedEntityRemoval	NBC	**78.8%**	73.9%	74.1%	82.2%	82.1%
	SVM	79.4%	72.0%	80.9%	85.7%	78.5%
StopwordRemoval	NBC	77.7%	70.0%	79.3%	84.4%	76.7%
	SVM	78.9%	71.2%	80.5%	85.4%	77.7%
AcronymExpansion	NBC	78.4%	72.0%	76.7%	83.3%	79.6%
	SVM	**80.5%**	73.1%	82.4%	86.8%	79.2%

Table 3.10: Results of the preprocessing investigation using unigram features preprocessed with the respective method. Bold faced accuracies denote absolute improvements (not significant) in comparison to the same classifier without the preprocessing.

None of the preprocessing steps yielded significant improvements compared to the averaged baseline of 79.46%. Most of the absolute accuracy values are even below the baseline. However, the bold faced accuracies are higher compared to the respective classifier without the preprocessing. They are neither significant though. Using a combination of those preprocessing techniques yielding absolute improvements also did not improve the accuracy significantly. Values of precision and recall per class are also very similar compared to the baseline.

There is only one conclusion that can be drawn here: The evaluated preprocessing techniques do not significantly improve accuracy. However, they also do not make it worse. Hence, when the size of the feature space becomes an issue, one could apply those preprocessing techniques to deal with the curse of dimensionality without being worried about the loss of accuracy. Thus, the common usage in current methods (see also section 2.3.3) seems odd to some extent since the authors do not mention any dimensionality problems. Yet, as it does not make things worse, no harm is done.

3.5 Combining Features

In literature the current methods for Twitter Sentiment Analysis claim to achieve better results by combining two different feature types. To verify or disprove this hypothesis, all combinations of two of the introduced features for NBC and SVM are evaluated and the results are presented in this section.

3.5.1 Naive Bayes Classifier

Combination of features for NBC are implemented using a two stage smoothing approach with Laplace smoothing for features and Jelinek-Mercer smoothing to combine classifiers (see also section 2.1.3).

The following equation generalizes the maximum likelihood estimation with Laplace smoothing, using feature type F with features f:

$$P_F(f|C_i) = \frac{TF(f, C_i) + 1}{\sum_{f' \in V} TF(f', C_i) + |V|} \; . \tag{3.7}$$

Harnessing this estimation, two classifiers $P_1(t|C_i)$ and $P_2(t|C_i)$ are trained using the respective feature type to compute the probability that a given tweet t is of class C_i. The two probabilities are then combined by Jelinek-Mercer smoothing which yields the smoothed probability that t is of class C_i:

$$P_\lambda(t|C_i) = (1 - \lambda)\; P_1(t|C_i) + \lambda\; P_2(t|C_i) \; . \tag{3.8}$$

After this smoothing step has been applied, the Naive Bayes Classifier proceeds as usual:

$$\text{classify}(t) = \arg\max_{C_i} P_\lambda(t|C_i) \; . \tag{3.9}$$

To approximate the optimal smoothing parameter value α*, a procedure similar to the one of the corpus size experiment (section 3.3) is used. For each combination of features, the α values are sampled from 0.1 to 0.9 in steps of 0.1. Zero and one do not have to be considered because those would result in the usage of just one of the classifiers. For each feature type, the minimum training size determined in section 3.3 is used as training data. The resulting classifiers are evaluated on the hand labeled training set. Finally, the effect of α is visualized in a graph. Figure 3.6 presents the graph for the combination of unigram and bigram features. Unigrams alone achieved 78.5% accuracy, bigrams 79.4%. The maximum accuracy for the combination is 81.0% using $\alpha = 0.5$. Hence, the result is an indication that combinations of features may yield overall benefits. Accuracy decreases when the weight shifts towards one of the classifiers while being at a plateau for $\alpha = 0.5, 0.6, 0.7$. However, just the improvement compared to unigram features is significant, the improvement compared to bigram features is not. Morever, just as expected, the difference to the general baseline also is not significant.

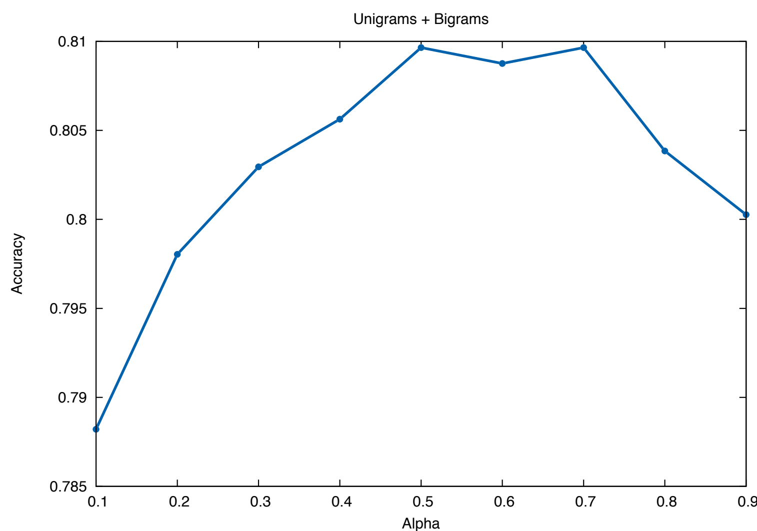

Figure 3.6: NBC created from a combination of unigram and bigram features. Accuracy is plotted against increasing values of the sampling parameter α.

This result is representative for combinations of features performing more or less on par with each other. There is an absolute increase of accuracy which is not significant compared to the baseline.

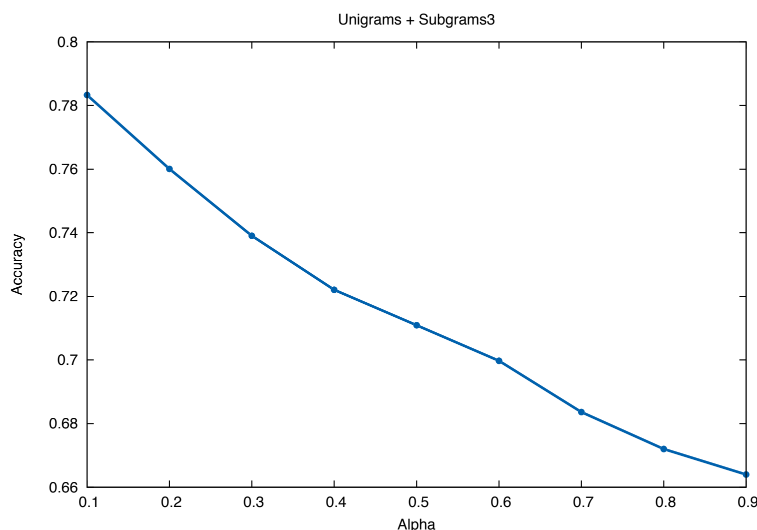

Figure 3.7: NBC created from a combination of unigram and subgrams3 features. Accuracy is plotted against increasing values of the sampling parameter α.

The behavior of the classifier when combining a good and a bad performing feature is illustrated in figure 3.7. It combines unigrams (78.5%) and subgrams3 (65%). With

increasing α, which means that it assigns more weight to the subgrams, the performance degrades. The best α for this combination is 0.1, assigning the maximum possible weight to the unigrams. Even though subgrams3 achieved the best values for positive recall and negative precision (see table 3.9), this feature does not seem to yield any improvements when used with two stage smoothing.

Table 3.11 sums up the results for all combinations investigated.

Features Combination	α^*	Accuracy	Prec+	Rec+	Prec-	Rec-
Unigrams+Bigrams	0.5	81.0%	76.2%	77.3%	84.3%	83.5%
Unigrams+Trigrams	0.6	80.3%	74.8%	77.5%	84.2%	82.2%
Unigrams+Subgrams3	0.1	78.3%	70.1%	81.5%	85.8%	76.1%
Unigrams+Subgrams4	0.1	79.0%	71.0%	81.4%	85.9%	77.3%
Unigrams+Pos	0.2	78.9%	71.5%	79.8%	85.0%	78.2%
Bigrams+Trigrams	0.5	79.9%	74.9%	76.3%	83.5%	82.5%
Bigrams+Subgrams3	0.1	79.8%	73.1%	79.7%	85.2%	79.9%
Bigrams+Subgrams4	0.2	80.5%	73.4%	81.5%	86.3%	79.7%
Bigrams+Pos	0.4	80.0%	72.8%	81.3%	86.1%	79.1%
Subgrams3+Subgrams4	0.9	**73.9%**	62.5%	89.7%	89.9%	63.1%
Subgrams3+Pos	0.1	**65.1%**	54.2%	92.4%	89.9%	46.4%
Subgrams4+Pos	0.1	**74.6%**	63.2%	89.9%	90.3%	64.2%

Table 3.11: Results for the evaluation of feature combinations for NBC. The column α^* denotes the parameter for the smoothing which produced this best result. Bold faced accuracies denote significant differences to the baseline.

Only three combinations (subgram features with each other, and subgrams with POS tags) resulted in significant differences from the baseline. Unfortunately, those differences are significantly worse. All other combinations performed on par with the baseline. However, some of them resulted in an absolute increase of accuracy.

To sum up, the combination of features with Naive Bayes Classifiers using two stage smoothing does not yield significant improvements over the baseline. Nevertheless, a combination of unigrams and bigrams performs significantly better than unigrams only, but it is statistically indistinguishable from bigrams only. For this reason, only bigrams or a combination of unigrams and bigrams should be used. Other combinations do not yield any improvements.

3.5.2 Support Vector Machine

To combine different kinds of features for SVMs, the respective feature vectors are combined. The feature space is extended by appending the second feature vector to the first one. For example, the unigram feature space has dimension n, the bigram feature space is of dimension m. Then the bigram features would be appended to the unigram features, resulting in a feature space of size $n+m$. Thus, when combining certain feature spaces, like trigrams and bigrams, the dimension could easily get out of hand. Various term weighting schemes could be applied additionally, see Joachims (2002) for details. Due to the this work being focussed on the basic methods, no weighting schemes are

used here. However, all combinations that could be computed have been evaluated. The procedure is the same as for the NBC. SVMs are trained with the obtained training size for the respective feature and evaluated on the hand labeled test set. Table 3.12 presents the results.

Features Combination	Accuracy	Prec+	Rec+	Prec-	Rec-
Unigrams+Bigrams	**82.6%**	74.8%	86.2%	89.4%	80.1%
Unigrams+Trigrams	**82.3%**	75.0%	84.6%	88.4%	80.6%
Unigrams+Subgrams3	79.5%	72.6%	79.9%	85.2%	79.3%
Unigrams+Subgrams4	79.5%	74.0%	76.4%	83.4%	81.6%
Subgrams3+POS	77.7%	72.3%	73.4%	81.6%	80.7%
Subgrams4+POS	77.8%	72.6%	73.0%	81.4%	81.1%
Unigrams+POS	79.8%	74.0%	77.4%	84.0%	81.4%
Bigrams+POS	78.9%	72.5%	77.3%	83.7%	80.0%
Bigrams+Subgrams3	77.7%	72.4%	73.1%	81.4%	80.9%
Bigrams+Subgrams4	78.6%	74.2%	72.9%	81.6%	82.6%
Trigrams+Subgrams3	**74.6%**	68.4%	69.9%	79.1%	77.9%
Trigrams+Subgrams4	**75.8%**	70.8%	69.0%	79.1%	80.5%
Trigrams+POS	**75.7%**	69.4%	72.2%	80.4%	78.2%

Table 3.12: Results for the evaluation of feature combinations for SVM. Bold faced accuracies denote significant differences to the baseline.

Bold faced accuracies indicate significant differences in comparison to the established baseline. Unigrams, combined with rather bigrams or trigrams yield significant improvements up to 82.6% while they are statistically indistinguishable from each other. However, the feature space and hence also the computation time is reasonably smaller for unigrams+bigrams. Thus one should favor this feature combination over unigrams+trigrams. Combinations of trigrams with subgrams or POS tags yield significantly worse results than the baseline. The rest of combinations performs neither significantly worse nor significantly better.

To summarize, combining features for SVMs does not only yield an absolute increase in accuracy but also significantly improves it compared to the baseline.

3.6 Conclusions

As as result of the performance investigation one can draw various interesting conclusions. Firstly, an analysis of how well features discriminate between classes can not necessarily be transferred to the performance of a classifier using those features. For example, POS tags seemed to discriminate the classes fairly well, according to the $\mathcal{X}^2$ analysis in section 3.1.4. However, classifiers using POS tag features did not perform very well.

Secondly, the commonly used preprocessing techniques like entity removal/replacement, spelling correction, acronym expansion, stemming, lemmatization and stop word removal did not result in any significant improvements. Nevertheless, they also did not

make the accuracy significantly smaller while still reducing the size of the feature space. Therefore, they may still be useful when high dimensions are becoming a problem for reasons of limitations in memory or computation time.

Finally, most of the proposed combinations of features also do not yield any significant improvements over the established baseline. Only combining unigrams and bigrams/trigrams to a feature space for SVMs yielded a significant improvement and achieved up to 82.6% accuracy. As far as I am concerned, the only current method significantly outperforming this is the one of Bakliwal et al. (2012) which achieved 88% accuracy on the testsets from Go et al. (2009) and Bora (2012). They have been using handcrafted sentiment feature vectors for SVMs, which have not been considered for this thesis because they have not been made public entirely. However, as they did not make any statements of the quality of the test dataset their results have to be interpreted with caution. Therefore, only a direct comparison on the same dataset can reveal if the difference is actually significant.

The major takeaway of this evaluation is the following: Many of the proposed methods do not yield any significant improvements over the canonical baseline while others, which have been discarded by many researches, do so. Support Vector Machines using a combination of unigrams and bigrams are the best performing standard classifier according to the investigation performed in this thesis.

4 Implementation of a Real Time Sentiment Tracking Application

In this chapter an implementation of a real time sentiment tracking application is presented. As the main focus of this work is the performance investigation in chapter 3, only a proof-of-concept application is provided. This application should be seen as a basic suggestion on how one should apply the obtained classifier to create a useful prototype and not as a full-fledged product that could be published right away.

After introducing the basic requirements, an overview of the architecture and the tools used is given. The next sections describe in-depth how exactly the specified features are implemented with the tools introduced. Finally, some conclusions regarding the development process and the tools used are drawn.

4.1 Requirements

First of all, the application should be web based. A web based application is accessible by any operating system which can run an arbitrary browser. Moreover, the complex and time consuming computations are performed on the server side. As a user, one just has to visit a web site to make use of the service. However, for the purpose of this thesis the application is implemented as single user application without user management. Thus, access control has to be taken care of separately. This can be done by making a simple HTTP authentication mandatory or by restricting the access to the application to particular networks.

Next, one should be able to define somehow which tweets' sentiment one would like to track on twitter. For this purpose a so called **entity** has to be defined by giving a list of relevant keywords. For example, when one would like to track the sentiment towards the company *Apple*, those keyword list could be something like *apple, ipad, iphone, mac, imac, osx*. Tweets containing one of the keywords are then assigned to the defining entity. This entities should be manageable through the web interface.

The processing and collection of the tweets should happen in the background automatically. Once the entities have been defined, the collection process should listen to the Twitter Streaming API[1] with regard to the relevant keywords. This process should also be scalable to some extent. Depending on the keywords given, there could be hundreds of thousands of tweets per minute which have to be handled by the application.

After after having collected relevant tweets for the defined entities, one is interested in visualizing their sentiment. The sentiment should be presented as a time series, indicating the ratio of positive and negative tweets over time. The time frame and the

[1] `https://dev.twitter.com/docs/streaming-apis`

resolution for the graph should be chosen by the user. If, for example, a company does a live event where they present new products, they may be interested in the sentiment at the exact time the product is mentioned in their presentation. Hence, they have to be able to choose a very high resolution like ten seconds. However, when they want to get informed about the sentiment towards the product five weeks after the launch, a proper resolution for this would be days or even weeks. Thus, the tweets' sentiment has to be aggregated over given time frames and resolutions dynamically.

Now that users can browse the sentiment towards their entities, they most likely want to explore the collected tweets to get an idea what caused the presented sentiment results. If, for example, the sentiment towards an entity on a particular day was bad, one would like to be able to see the tweets from that day. After browsing those one may get an idea which keywords are relevant for that day's sentiment. As the user does not want to browse all the tweets, he wants to be able to filter the presented tweets by providing a search query. Moreover, it would be nice if the sentiment graph now also adapted to the search and displayed the sentiment of the filtered tweets.

Finally, the sentiment of the various entities should be comparable in some kind of dashboard. For example, when tracking an election, one would like to see the sentiments towards multiple political parties in comparison, not each one separately. However, it is also crucial that entities can be hidden when there are too many of them.

Summarizing the requirements, a web application is desirable in which entities can be defined by a keyword list. Tweets relevant for those keywords should be obtained and analyzed. The results should be presentable with regard to a given time frame and resolution. Tweets should also be browsable, and should be filterable using a full-text query. The sentiment graph should dynamically adapt to the filtering and present the sentiment for filtered tweets. Finally, the sentiment graphs of all entities should be visualized in conjunction at a dashboard page.

4.2 Overview of the Architecture and the Tools Used

This section presents an overview of the various components used to implement the features specified in the preceding section 4.1. Additionally, it provides information on how those components work together. Figure 4.1 illustrates all components and their connections.

The basis of the application is formed by a Ruby on Rails[2] (Rails) application. Rails is a framework written in the Ruby programming language[3]. David Heinemeier Hansson, author of Rails, felt the need to develop a framework which makes the creation of web applications as easy as possible. He achieved this by applying various design patterns like **Model-View-Controller** (MVC), **Convention over Configuration** (CoC) or **Don't Repeat Yourself** (DRY)[4] and exploiting the dynamic nature of Ruby. As a result, working with Rails is really comfortable and productive. Due to its ability to

[2] `http://rubyonrails.org`
[3] `https://www.ruby-lang.org/`
[4] For further details see Haldenwang (2011).

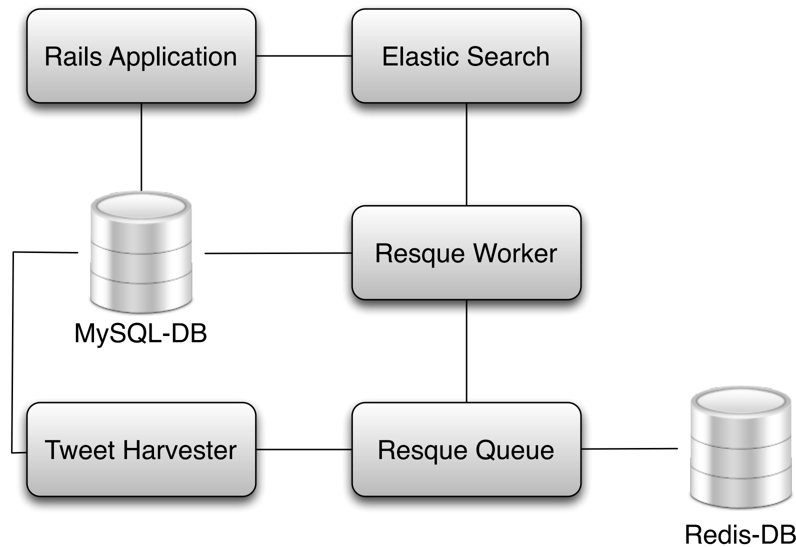

Figure 4.1: Overview of the applications components and their interactions.

enable developers to create stable working prototypes in relatively short time, it has become the status quo in the German web startup scene.

Within the architecture, the Rails application responsibilities are to provide an interface to manage entities, store them in a **MySQL** database[5] and retrieve them when needed. Moreover, it handles the storage of harvested tweets and keywords, also using the MySQL database. It also includes the presentation layer which brings the gathered data to the user. Tweets can be browsed in dynamic HTML tables created with the **Datatables** library[6]. The overall layout is designed using **Twitter Bootstrap**[7]. To visualize the sentiment graphs, the chart framework **Highcharts**[8] is made use of. Details are described in later sections. Figure 4.2 presents the underlying data model of the Rails application.

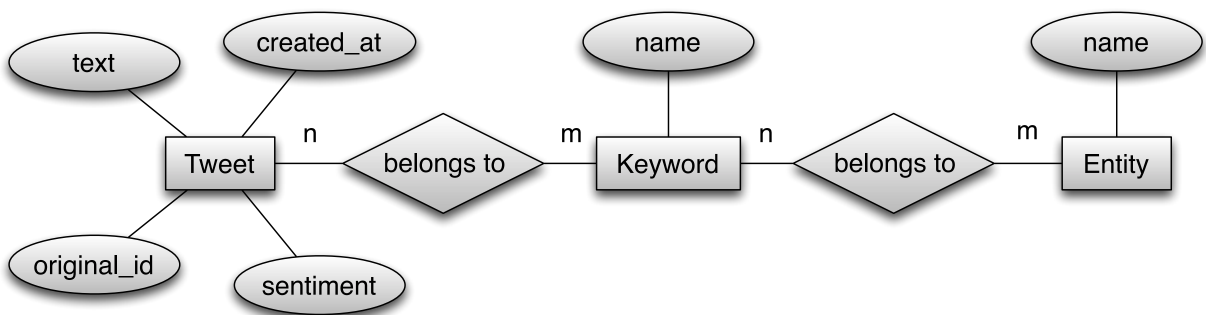

Figure 4.2: Datamodel of the Rails application in ER notation. Primary keys are artificial *id* attributes, which are not shown here. Some semi-relevant tweet meta data attributes like *retweet_count* are also left out.

[5]http://www.mysql.com
[6]https://datatables.net
[7]http://getbootstrap.com
[8]http://www.highcharts.com

Acquisition of tweets from the Twitter Streaming API is done by a component called **tweet harvester**. This process queries the MySQL database to obtain the list of relevant keywords. After that, it starts listening to the tweet stream for tweets containing those keywords. Since the developer documentation from twitters Streaming API suggests that the process obtaining the tweets should not be the process analyzing them, this is done by a separate component. Doing the processing within the harvester would result in it not to be able to fetch all tweets which are send through the stream, which would end in twitter terminating it. Therefore, the harvester pushes the obtained tweets into a queue for further processing. To realize such a queue, the Ruby library **Resque**[9] is used. Resque uses a very performant in-memory Redis[10] database to store queued data. Redis has a native data type for atomic queues which makes this fairly easy, even if multiple processes work with it in parallel. More details are to be shown in the following sections.

Now that the tweets have been obtained and are waiting in the queue for processing, a **Resque worker** process can poll them to do this. It transforms the raw data received from twitter to the data format expected by the rails application and inserts it into the MySQL database. Along with this, the tweet also is indexed using **Elastic Search**[11]. Elastic Search is a full-text search engine which is used to realize the filtering feature.

Since the basic components and their interconnections are known now, the following sections highlight the implementation of each feature in greater detail, showing how exactly the components work.

4.3 Entity Management

A simple interface has been implemented for the creation of entities. A screen shot is shown in figure 4.3. This could be done by using basic Rails features only. The user enters the name of the entity, a comma separated list of keywords and then just presses the *Create Entity* button to start tracking the sentiment.

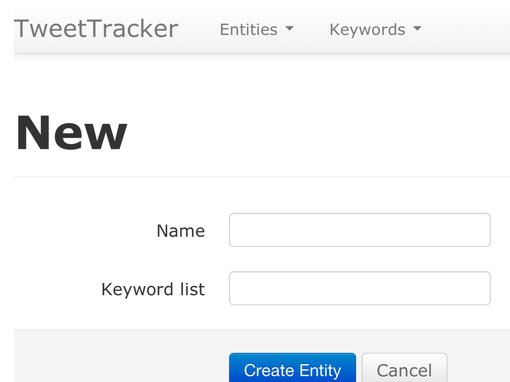

Figure 4.3: Screenshot of the interface to create a new entity.

[9] https://github.com/resque/resque
[10] http://redis.io
[11] http://www.elasticsearch.org

Looking at the underlying data model, presented in figure 4.2, one notices that keywords are their own entity and not simply an attribute of the model `Entity`. Hence, the application has to make sure that the *many-to-many* relation is handled accordingly and that necessary instances of `Keyword` are created with the respective name. The complex part here is not to create the entity, but to edit and update the list of keywords correctly. This problem, one comes across once in a while, can be approached by treating the old and the new keywords as sets. Let A be the new set of new keywords and B be the set of keywords present before the update. The simplest action has to be performed for the set $A \cap B$. Those are keywords that have been in the old list and still are in the new list. Therefore, nothing has to be done for them. If $A \setminus B$ is the set of keywords that has to be added, it has to be checked if an instance of `Keyword` with the given name exists. If there already is one, only the relation has to be added, if there is not, it has to be created first. The set $B \setminus A$ includes the keywords which have been there before but should be removed now since they are no longer in the list. Listing 3 presents the resulting Ruby code from the model class `Entity`.

```ruby
class Entity < ActiveRecord::Base
  has_and_belongs_to_many :keywords

  def keyword_list
    keywords.map(&:name).join(", ")
  end

  def keyword_list=(list)
    transaction do
      new_keyword_list = list.to_s.split(",").map(&:strip).
                                             map(&:downcase)
      old_keyword_list = keywords.map(&:name) || []

      to_add = new_keyword_list - old_keyword_list # A \ B
      to_add.each do |keyword|
        keywords << Keyword.where(name: keyword).first_or_create
      end

      to_remove = old_keyword_list - new_keyword_list # B \ A
      to_remove.each do |keyword|
        keywords.delete Keyword.where(name: keyword).first
      end
      # nothing to do for the intersection of A and B
    end
  end
end
```

Listing 3: Code from the entity model, illustrating the handling of keyword lists.

In order to prevent undesired anomalies from occurring, the whole procedure is wrapped into a transaction. All other necessary steps are handled by the rails application au-

tomatically. The field from the HTML form is named *keyword_list* internally. Hence, Rails uses the method `keyword_list` to populate the field with data when updating the entity and it uses the method `keyword_list=(list)` to process the value sent from the form. By overwriting those two methods with the desired behavior, the handling of the keyword list is easily implemented.

4.4 Harvesting and Processing Tweets

Tweets are collected using Twitter's **public streaming API**[12]. They are providing a freely available but limited stream, yielding a random portion of all tweets currently tweeted. Only one connection can be made per user account. How many tweets one gets is mostly limited by how fast one can read from the stream. It rarely happens that no more tweets can be obtained due to the limitation. Since one is only interested in getting tweets including relevant keywords for the created entities, the filtered stream[13] is used. It can be filtered by keywords, authors or locations. For this application only the keyword filter is harnessed. Figure 4.4 illustrates the tweet processing pipeline used in this application.

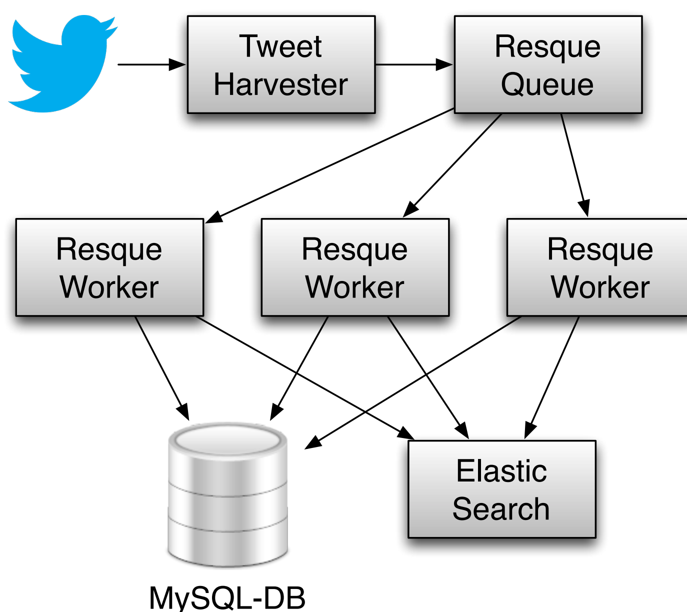

Figure 4.4: Data flow of the tweet processing pipeline.

Through Twitter's streaming API, the tweet harvester obtains the relevant tweets. This is done with the Ruby gem **Tweetstream**[14]. The relevant information is pulled out, encoded as JSON and handed over to the Resque queue where it is stored until a worker polls it, processes the data and inserts it into the MySQL-DB and the Elastic Search full-text search engine. Listing 4 illustrates the core of the tweet harvester process.

[12] `https://dev.twitter.com/docs/streaming-apis/streams/public`
[13] `https://dev.twitter.com/docs/api/1.1/post/statuses/filter`
[14] `https://github.com/tweetstream/tweetstream`

```ruby
daemon = TweetStream::Daemon.new('tracker')

buffer = []
buffer_size = 20

keywords = Keyword.active_keywords.join(",")

daemon.track(keywords) do |tweet|
  data = {}

  data[:created_at] = tweet.created_at
  data[:text] = tweet.full_text
  data[:original_id] = tweet.id
  data[:language] = tweet.text.language

  buffer << data

  if buffer.size >= buffer_size
    Resque.enqueue(TweetProcessingJob, buffer.to_json)
    buffer.clear
  end

end
```

Listing 4: Core of the tweet harvester process. Irrelevant configuration is left out here.

One very nice feature of Tweetstreamer is that it comes with daemon mode included. This makes it particularly easy to start the tweet harvester in the background, restart it if necessary or check if it is running. The class `TweetStream::Daemon` makes this possible, it is instantiated in line one. The run management can be done via commandline:

`tweet_harvester {start, stop, restart, run, status}`.

At first, the daemon object is created with the name *tracker*. Next, a buffer of size 20 is initialized. The buffering is necessary so that not every single tweet is put into the queue separately, but rather a package of multiple tweets. This buffering procedure minimizes the IO overhead occurring while communicating with the queue. Queueing just one tweet at a time is just not beneficial, because the IO overhead of putting it into the queue and polling it back out is bigger than the actual processing time.

After obtaining the keywords from the database using the `Keyword` model class of the Rails application, the tracking process can be started. To do this, one has to call the method `track` on the daemon object, handing it the list of keywords as a parameter along with a code block that is used to process the tweet. The given code block is called for each tweet which is received from the stream. It pulls out the relevant data first, then adds it to the buffer. Finally, it is checked whether the buffer is full, and if so,

its content is JSON encoded and handed over to the Resque queue, adressed at the job class `TweetProcessingJob`, while also clearing the buffer. No further processing is done here. Twitter suggests not to do any processing in the harvesting process, since that would slow down the rate at which the tweets are received and may result in Twitter closing the connection due to being too slow.

Processing of the tweets is done in Resque worker processes. To handle large traffic, an arbitrarily large number of workers can be started in parallel. They do not even have to be at the same machine. All they need is access to the Redis server containing the queue data. The whole process of polling data from the queue is handled by Resque. One just has to implement a class handling the given data in a class method called `perform`.

The method `perform` receives the JSON encoded buffer put into the queue by the harvester. First of all, the data is decoded, then, for each tweet of the package, its data is saved to the database using the applications model `Tweet` by handing over the decoded data to its constructor. Due to its simplicity, the code is not shown here. The actual processing is done with callbacks in the model class, as presented in listing 5.

```
1  class Tweet < ActiveRecord::Base
2    after_create :connect_with_relevant_keywords!
3    before_save :set_sentiment!, on: :create
4
5    private
6      def connect_with_relevant_keywords!
7        words = text.downcase.words
8
9        Keyword.all.each do |keyword|
10         if words.include?(keyword.name)
11           keywords << keyword
12         end
13       end
14     end
15
16     def set_sentiment!
17       sentiment = SentimentClassifier.classify(text)
18     end
19 end
```

Listing 5: Callbacks of the tweet used to assign a sentiment and connect it to the relevant keywords.

Callbacks are part of the `ActiveRecord` object life cycle. They allow the developer to inject certain actions into specific places in the life cycle. Before a tweet is saved, but only on its creation, the sentiment classification has to take part. Hence, in line four a `before_save` callback is issued, telling the model class to call the method `set_sentiment!` before saving. This method harnesses an implementation of the sentiment classifier obtained in the preceding chapter 3 to set the sentiment for the tweet.

To create a connection between the tweet and the keywords it is relevant for, it has to be saved first, so that its primary key is generated and can be used as a foreign key for the connection. Hence, the method `connect_with_relevant_keywords!` is invoked as an `after_create` callback. It compares the tweet's text with all keywords stored in the database and creates a connection when there is a match. A very similar mechanism is used to add the tweet to the index of Elastic Search. This is explained in more detail in section 4.5 about the full-text search engine.

4.5 Browsing an Entity's Tweets with Full-Text Search

In this section it is explained how one could easily make tweets filterable using full-text search. At first, the client side is discussed, next, it is explained how tweets are indexed, and finally, an illustration is given on how those two components are connected.

4.5.1 Presenting Tweets with Datatables

Figure 4.5: Screen shot of the provided interface to browse an entity's tweets.

Starting with the client side, figure 4.5 presents a screen shot of an entity's tweet browsing interface. The entity currently looked at is the company *Apple*. The keywords are also shown next to the table, but they have been left out here for reasons of space.

Per default, the table shows all available tweets, using **pagination**. Pagination denotes a method to only show a given number of tweets per page and provide links to the other pages. The page buttons are located at the bottom of the table and are also not included in the screen shot due to limitations of space. However, the number of tweets per page can be selected right at the top.

As a first kind of filtering, the user can provide a time frame through the input fields labeled *From* and *To*. When clicking one of the input fields, a comfortable graphical interface to pick a date and time pops out, which has been created with the **jQuery UI**[15] add on **Timepicker**[16]. Within the screen shot the timepicker for *To* is popped out.

Finally, the user can provide an arbitrary search query using the input field labeled *Search* to filter tweets he is interested in. The search makes use of the tweet's author's name, its text and even its sentiment. If somebody wants to filter all tweets of the author *Cindy*, including the word *Smurf* and with positive sentiment, the query *Cindy Smurf positive* returns exactly those.

The attentive reader might have noticed the lack of a submit button. Using the library **Datatables**[17], integrated with the Ruby gem **jquery-datatables-rails**[18], this is not necessary. Datatables can be told to reload itself when needed. Such an integration of datatables on the client side is also fairly easy. First of all, one has to create a HTML table including only the table header. In the next step, some javascript is attached to make it a datatable. Listing 6 shows the most relevant code for this.

```
1  var oTable = $('.dataTable').dataTable({
2    "sDom": "<'row'<'span9'l><'toolbar span5'>" +
3            "<'span4'f>r>t<'row'<'span4'i><'span5'p>>",
4    "sPaginationType": "bootstrap",
5    "bServerSide": true,
6    "sAjaxSource": $('#tweets').data('source'),
7    "fnServerParams": function ( aoData ) {
8          aoData.push( { "name": "entity_id",
9                         "value": $('#tweets').data('entity_id') } );
10         aoData.push( { "name": "from_datetime",
11                        "value":  datetimeFrom } );
12         aoData.push( { "name": "to_datetime",
13                        "value":  datetimeTo } );
14   }
15 });
```

Listing 6: Relevant code for the Datatables integration.

[15] http://jqueryui.com
[16] http://trentrichardson.com/examples/timepicker/
[17] https://datatables.net
[18] git://github.com/rweng/jquery-datatables-rails.git

Datatables is integrated by selecting the DOM[19] object where the table should be rendered to. The `table` object has been assigned the CSS class `dataTable` to do this. Next, the method `dataTable()` is called on it with the desired configuration parameters. The parameters `sDom` and `sPaginationType` contain configuration for the table's appearance. With `bServerSide`, the table is told not to do any processing by itself, and just to send queries to the server. Using `sAjaxSource` one could specify the URL to which the query is sent. Here, this is read right from an HTML5 data attribute of the table itself. Finally, one has to add the parameters to the query sent to the server, which are not handled per default. The query term and the pagination are native components of Datatables and hence have not be treated separately. However, the time frame and the *id* of the entity the user is currently looking at are not. The entity's *id* can also be retrieved from the table's data attributes. The time frame is acquired from the variables `datetimeFrom` and `datetimeTo`. Those are set by Timepicker when the user changes one of the time frame fields. Moreover, Timepicker also notifies Datatables of changes so it can reload, using a callback. Since the query term and the pagination values are native components of Datatables, the table reloads automatically once their values change.

4.5.2 Indexing and Retrieving Tweets with Elastic Search

Full-text searching with MySQL is not very efficient in general, especially when using InnoDB[20] as storage engine, as Rails does per default. Therefore, a tool developed especially to perform efficient full-text searching is used: Elastic Search[21]. On its website, it is described as a *"flexible and powerful open source, distributed real-time search and analytics engine for the cloud"*. It has been applied successfully by large web companies like **Stack Overflow**[22], **StumbleUpon**[23] or **SoundCloud**[24]. Respective case studies are provided at the Elastic Search website[25]. Moreover, popular Rails cloud hosting providers like **Heroku**[26] offer built-in packages for Elastic Search. Hence, it seems to be a widely accepted tool to perform efficient full-text search, and thus is used in this work.

Elastic Search is built upon **Apache Lucene**[27] and provides a variety of features. The system can easily be configured to run on a cluster of multiple machines and it automatically handles distribution and failure management. This makes the system overall performing well, allowing real time computations and analysis. Moreover, it is document-oriented, which means that no strict scheme for the data has to be given. Everything is stored as JSON objects. Furthermore, the interactions are performed with a comfortable REST interface, also using JSON as communication language.

[19] http://www.w3.org/DOM/
[20] http://dev.mysql.com/doc/refman/5.0/en/innodb-storage-engine.html
[21] http://www.elasticsearch.org
[22] http://stackoverflow.com
[23] http://www.stumbleupon.com
[24] https://soundcloud.com
[25] http://www.elasticsearch.org/case-studies/
[26] http://www.heroku.com
[27] http://lucene.apache.org/core/

However, for this work the REST API is not used directly. Fortunately, there is a Ruby gem called **Tire**[28] which integrates the Rails model classes with Elastic Search. One benefit of this integration is that as a developer one does not have to take care of any REST communication, query building or index management. Listing 7 presents a simplified version of the configuration for the model class `Tweet`.

```
1  class Tweet < ActiveRecord::Base
2
3    include Tire::Model::Search
4    include Tire::Model::Callbacks
5
6    mapping do
7      indexes :text
8      indexes :entities_ids, as: 'entities_ids'
9      indexes :created_at, type: 'date'
10     indexes :sentiment
11   end
12
13   def entities_ids
14     entities.map(&:id)
15   end
16 end
```

Listing 7: Relevant excerpt of the tire indexing code. Configuration to handle multiple languages with special analyzers is left out.

First of all, some model specific modules are included in lines three and four. Those enhance the class `Tweet` with various functionalities needed to interact with Elastic Search. For example, the module `Tire::Model::Callbacks` extends the class with `after_save` callbacks similar to those presented in section 4.4, which automatically handle the indexing when an instance of `Tweet` is created. Thus, the indexing actually takes place in the Resque worker, where the `Tweet` instances are created.

Moreover, the class method `mapping` is provided, which accepts a block argument containing the indexing configuration. The statement `indexes :text` results in the model attribute `text` being indexed in a field of the same name. For more complex data that is not a direct model attribute, one can provide the name of a method as second argument to the method `indexes`. The return value of the method is indexed. Line eight includes such a use case. To be able to only retrieve tweets which are connected to a certain entity, one needs to index their *ids*. Since the entities' *ids* are no attributes of `Tweet`, Tire is told to use the method `entities_ids` to retrieve them. The method, defined in line 13, simply constructs an array including each relevant entity's *id*. Whenever the tweet is updated, the method is called again to retain consistency. Furthermore, a data type can be defined if necessary. For example, one would like to index the tweet's creation timestamp as `date` and not as string. This is configured in line nine.

[28] `https://github.com/karmi/tire`

Without having to deal with any REST or JSON, the index is automatically created and updated at correct times preserving consistency between the MySQL database and Elastic Search. For more information on the multitude of features, the interested reader is invited to take a closer look at the Tire gem documentation[29].

Querying the index is not very complicated either. Listing 8 presents a simplified version of the query used to browse tweets. The real query is embedded into a class method on `Tweet`, and also handles missing or wrong parameters by only applying filters actually given. To keep the example simple, those steps are left out here.

```
Tweet.search do
  query do
    boolean do
      must { string query_string , default_operator: "AND" }
      must { term :sentiment     , sentiment              }
      must { range :created_at   , from: start_date,
                                   to: end_date           }
      must { terms :entities_ids , entities_ids           }
    end
  end
end
```

Listing 8: Illustration of the query API of Tire.

A search is performed by calling the method `search` on the model class. This method has been made available by inclusion of the Tire modules. As a parameter it expects a block which consists of the code constructing the actual query. If one wants to receive only those tweets which fulfill all criteria, one has to wrap the filters into a `boolean` block and each criterion into a `must` block. As a result, they are concatenated with a logical *and*. The first filter in line four is of the kind `string`. It is passed the query string the user entered into the search field. If the query term consists of multiple words, they should also be concatenated with a logical *and*. Next, a filter of the type `term` is applied to the sentiment. Term fields are not analyzed in any manner, they just match for identity. To only retrieve tweets from the desired time frame, one can use a range filter, passing it the name of the date field and the start and end date of the time frame. Finally, to scope the search with regard to the entity one is currently interested in, a `terms` filter is used. It works just like `term` but handles lists of multiple terms since a tweet can be relevant for multiple entities.

```
results = Tweet.search { "left out" }
first_result   = results.first
tweet_text     = first_result.text
tweet_datetime = first_result.created_at
```

Listing 9: Example code on how to access the results of a tire search.

[29] http://karmi.github.io/tire/

The results can easily be accessed using accessor methods which are named like the indexed attributes. Listing 9 provides an example.

4.5.3 Connecting Datatables and Elastic Search Using the Presenter Pattern

The default for handling HTTP requests in Rails is to map every URL to a particular controller action. Within the controller action, the requested data is acquired from the database using model classes, and it is then returned to the client in the desired format. However, in some cases relatively complex transformations of the data are necessary. Providing data for Datatables is such a case. Request parameters have a special format that has to be handled explicitly. Additionally, Datatables expects a certain response format to be able to process the data correctly. While all necessary transformations could be done in the controller action, this is strongly discouraged. A controller action should basically just delegate the calls to gather the data somewhere else and respect the single responsibility paradigm. Hence, a controller should just control the data flow and do no direct data processing. The actual code from the application is presented in listing 10. As suggested, the action `index` does nothing other than handling the different response formats while delegating the request's processing and response generation to an instance of `TweetsDatatable`.

```
1  class TweetsController < ApplicationController
2    def index
3      respond_to do |format|
4        format.html
5        format.json do
6          render json: TweetsDatatable.new(view_context)
7        end
8      end
9    end
10 end
```

Listing 10: Illustration of the controller code which processes the Datatables request.

Since the code to generate the required format is basically presentation logic, it does not fit into the model class. Thus, it is extracted to a new presenter class. As there is no view for creating the JSON data expected by Datatables, the **presenter pattern**[30] is made use of. The basic idea of the presenter pattern is to create intermediate objects between view and controller to retain clarity. As shown in line six, the presenter class `TweetsDatatable` receives the controller's current `view_context` to get access to the request object, and to the available view helper methods. As a result, it gets both the capabilities of a view and partly of the controller. The methods `user_link`, `sentiment_icon` and `format_date` are not included due to lack of space. They just

[30] For a general introduction to presenters in Rails see: `http://railscasts.com/episodes/287-presenters-from-scratch`, for a Datatables specific version see: `http://railscasts.com/episodes/340-datatables`.

construct the HTML code one would expect considering the method's name. Moreover, parameter sanitization, such as preventing cross site scripting attacks or SQL injections, is also left out but it is performed in the application of course.

```
1  class TweetsDatatable
2    delegate :params, :h, :link_to, # [...]
3      to: :@view
4
5    def initialize(view)
6      @view = view
7    end
8
9    def as_json(options = {})
10     {
11       sEcho: params[:sEcho].to_i,
12       iTotalRecords: tweets.total,
13       iTotalDisplayRecords: tweets.total,
14       aaData: data
15     }
16   end
17
18 private
19   def data
20     tweets.map do |tweet|
21       [
22         user_link(tweet.user_name, tweet.user_id),
23         tweet.text,
24         sentiment_icon(tweet.sentiment),
25         format_date(tweet.created_at)
26       ]
27     end
28   end
29
30   def tweets
31     @tweets ||= Tweet.search_tweets(
32       query_string: params[:sSearch],
33       page:         params[:iDisplayStart],
34       per_page:     params[:iDisplayLength],
35       entity_id:    params[:entity_id],
36       start_date:   params[:from_datetime],
37       end_date:     params[:to_datetime]
38     )
39   end
40 end
```

Listing 11: Simplified excerpt of the Datatables presenter class `TweetsDatatable`. Some methods used are left out due to lack of space.

A simplified excerpt of the presenter class `TweetsDatatable` is shown in listing 11.

First of all, the class has to store the controller's view context in an instance variable called `@view`. Lines two and three set up the delegation to the view context. As a result, if a method is not known to `TweetsDatatable` but can be delegated to the view context, it will be invoked there instead. For reasons of space, some delegated methods are left out.

The method `as_json` is called by the controller when trying to render the object in the JSON format. Hence, it has to return the desired JSON representation for Datatables. It has to include an only internally used field named `sEcho`, the number of records to be displayed and the actual data to fill the table with.

Generating the data is done with the private method `data`. This method creates a two dimensional array, the first dimension representing the tables rows, the second the columns. Note that the order for the second dimension matters, the order of data has to match the order of the columns in the table.

The tweets matching the search query are provided by the method `tweets`. To load the tweets, a query building method `search_tweets` (see section 4.5.2) is called, passing it the relevant parameters which have been sent by Datatables. Moreover, this method makes use of the **lazy loading** pattern, using the operator `||=`. If the instance variable `@tweets` exists, nothing will happen and it will just be returned. If it does not exist, the query will be sent, the result will be assigned to `@tweets` and finally the tweets are returned. Lazy loading prevents unnecessary loading and reloading of data by ensuring it is just loaded when it is really needed and cached afterwards for further usage.

While the example presented consists of just 40 lines of code, the real `TweetsDatatable` class includes about 80 lines of code. Having this code cluttering the controller in a non object oriented manner would make it hard to understand and maintain in the long run. That is why the usage of presenters is the design pattern of choice for use cases similar to the one illustrated here.

4.6 Visualizing the Entities' Sentiment

In this section, the process of visualizing the sentiment towards an entity is presented. To start with, the client side user interface is presented and it is shortly illustrated how it has been integrated into the application. Afterwards, the server side computation of the time series data is explained. Since the communication between the components works similarly to the one described in section 4.5.3, it is not discussed in detail again here.

4.6.1 Drawing Charts with Highcharts

The visualization of the sentiment towards an entity is done using the pure HTML5 and Javascript library Highcharts[31]. Various kinds of charts are supported, such as

[31] `http://www.highcharts.com`

splines, bar charts, pie charts, time series charts, box plots and many more. Customers of Highcharts include IBM, NASA, Siemens, HP, CBS, BBC and various other well known companies. The toolkit can be used without fee for non commercial applications but one has to purchase a license to use it for commercial purposes. Being based on HTML5/Javascript only is a huge benefit since no further plugins are needed and most current browsers can run Highcharts out of the box.

Figure 4.6 shows the sentiment chart for an entity. Right on top of the chart the grouping interval can be chosen. Grouping tweets in ten minute intervals, as it is the case in the figure, results in the sentiment being aggregated for all ten minute blocks in the total time frame. Put simply, each data point represents the sentiment of a ten minute interval of time. Whenever the user selects another resolution, the chart automatically reloads the data from the server right away. By hovering a data point, the user is presented a small layover including the exact starting time of the data points time interval and the exact polarity score. The chart is located right below the datatable introduced in the preceding section 4.5. Moreover, it also updates itself automatically whenever the datatable is updated due to filters applied by the user.

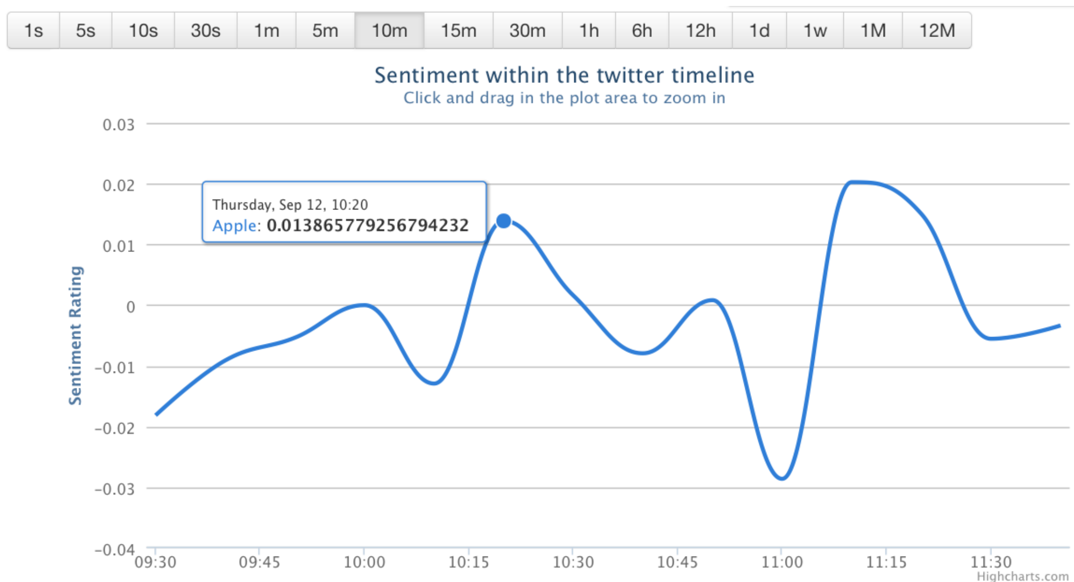

Figure 4.6: Sentiment towards an entity, visualized with Highcharts.

The actual polarity score $P_T \in [-1, 1]$ for a time interval T is computed as

$$P_T = \frac{t_T^+ - t_T^-}{t_T^+ + t_T^-}, \tag{4.1}$$

with t_T^+ being the number of positive tweets in the interval and t_T^- being the number of negative tweets in the interval. The polarity measure's sign intuitively reflects the overall sentiment. Positive polarity means positive sentiment, negative polarity means negative sentiment. Due to the normalization to the interval $[-1, 1]$, the chart retains its readability compared to visualizing the pure number of tweets. The number may differ strongly for various time intervals, which would drastically increase the scale of the charts and hence make the intervals with lower numbers of tweets unreadable.

A very similar chart is used for the dashboard view, where multiple entities' sentiments can be compared. Figure 4.7 presents a screen shot. The visualized polarity score is the same, along with the interface to choose the grouping interval. However, this chart needs its own input element for the time frame to be visualized, since this view does not contain a datatable from which those data can be read. Again, the date picker described in section 4.5.1 and shown in figure 4.5 is used. Moreover, it visualizes multiple entities at once. Below the chart, a legend illustrates which spline belongs to which entity. Note that entities can be faded out by clicking the legend. In the screen shot, for example, the entity *Apple* is faded out. Another click would fade it back in. Additionally, the layover now includes the polarity scores for all visualized entities, labeled with the entities' names.

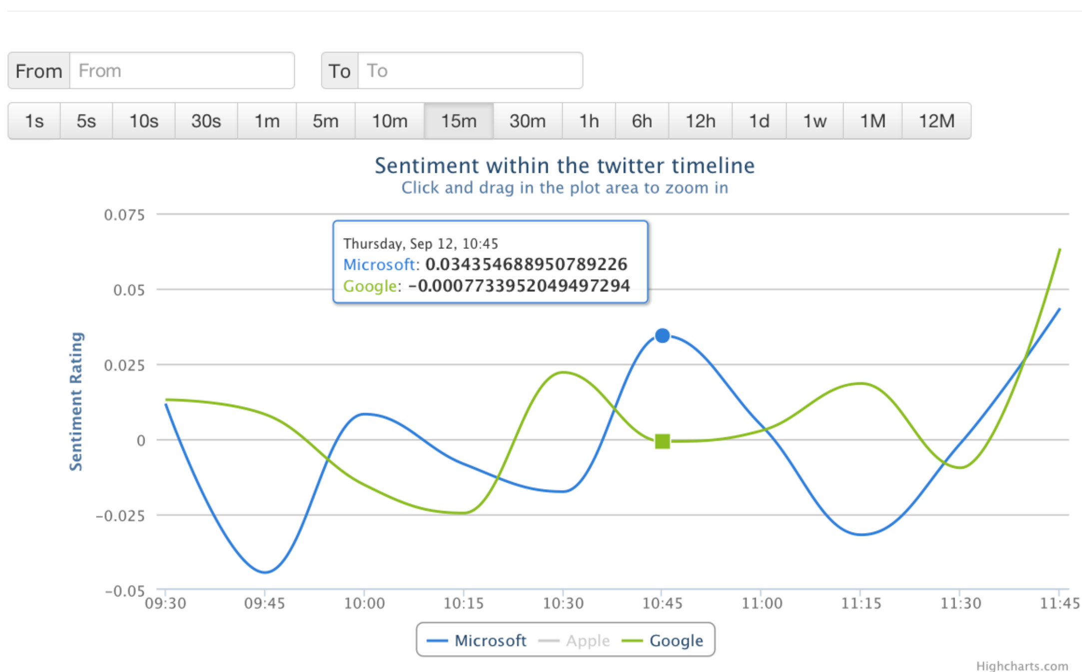

Figure 4.7: Chart to compare the sentiment of various entities.

Integrating Highcharts into the **Rails Asset Pipeline**[32] can be achieved easily with the Ruby gem **Highcharts Rails**[33]. The actual drawing of the chart is done in Javascript by telling Highcharts to create a chart using the specified options. The aforementioned options basically control the appearance of the chart, like labels for the axes, the type of chart or the data source. Configuring is done by creating a JSON object including the desired settings. Due to its length and as it provides no further insights, the configuration is not presented here.

However, loading of the data varies a little from the default behavior. Listing 12 presents the reload function. It uses the default API from jQuery[34] to perform asynchronous

[32] http://guides.rubyonrails.org/asset_pipeline.html
[33] https://github.com/PerfectlyNormal/highcharts-rails
[34] http://api.jquery.com/jQuery.ajax/

requests. Since a variety of parameters has to be sent and processed, the HTTP verb `POST` is used. Additionally the URL has to be given. The request parameters are very similar to those of Datatables. In order to scope the data for the entity currently looked at, in line six the entity's *id* is retrieved. The selection of the correct time frame is again done by using the variables `datetimeFrom` and `datetimeTo` which are provided by Timepicker. Next, the grouping interval is set using the variable `interval` which is set by the interval selection element whenever the user selects a new value. Finally, the query string has to be acquired from the Datatables search input field. After setting the `dataType` to JSON, a callback function is defined. This function is called after the request has been performed successfully. Within the callback function, the retrieved data is added to the configuration object `options`. At last, the chart is created by passing Highcharts the desired configuration including the data.

```
1  function reloadHighcharts(){
2      $.ajax({
3          type: "POST",
4          url: "/tweets/histogram.json",
5          data: {
6            entity_id: $('#tweets').data('entity_id'),
7            from_datetime: datetimeFrom,
8            to_datetime:   datetimeTo,
9            interval:      interval,
10           query_string:  $('div.dataTables_filter input').val()
11         },
12         dataType: 'json',
13         success: function(data){
14           options.series = data;
15           chart          = new Highcharts.Chart(options);
16         }
17     });
18 }
```

Listing 12: Function to load the data from the server and redraw the chart.

The complete reload code is wrapped by the method `reloadHighcharts`. This method is called on various occasions. First of all, it is invoked once the page has loaded to initially draw the chart. Additionally, the method is invoked whenever Datatables performs a reload to keep both components in sync. They should always display the same data. Finally, when the user changes the resolution of the grouping interval, another reload is performed. This does not affect the synchronization with Datatables because the data does not change. Just the aggregation across the time intervals changes.

Code for the dashboard chart is not presented here, as it looks very similar and would not provide any further insights.

4.6.2 Computing Time Series Data with Facet Searches

As a user can dynamically change the aggregation interval for the sentiment time series, it has to be computed dynamically. Thus, the computation has to be very efficient to keep the client side interface fluent. The canonical approach is to retrieve all relevant tweets, iterate over them and group them according to the given aggregation interval. However, there would be huge overhead here. Depending on the entity's keywords, the number of tweets may rise up to millions in short time. Therefore, retrieving all of them produces a lot of overhead since one is not interested in the tweets themselves but just in their count with respect to a given sentiment in a given interval of time. Fortunately, Elastic Search has a feature called **Facets**[35] available. While a full-text search engine in general is designed to quickly return a small number of documents matching the given query, facets also allow for online computation of aggregated data on the documents matching the query. The facet functionality is illustrated with the example given in the documentation of the Tire gem[36]. The example consists of an `Article` model class which has two attributes, a list of `tags` and a `title`. Listing 13 presents the creation and indexing of the data, using the Tire enhanced model class (not shown here).

```
1 Article.create(title: "One",   tags: ["ruby"]           )
2 Article.create(title: "Two",   tags: ["ruby", "python"] )
3 Article.create(title: "Three", tags: ["java"]           )
4 Article.create(title: "Four",  tags: ["ruby", "php"]    )
```

Listing 13: Creating the data for the facet example.

Now that there is some data to work with, the facet search can be performed. The code is shown in listing 14. First of all, one has to set a query string for the full-text search. Using the query `title:T*`, only articles with titles starting with a capital *T* will be returned. No further restrictions are made.

```
1 s = Article.search do
2   query { query_string "title:T*" }
3
4   facet 'tags' do
5     terms :tags
6   end
7 end
```

Listing 14: Code to perform a facet search.

Next, the facet is created using the method `facet`. To access the results later, one has to assign a name to the facet by passing it in as a first parameter. In this example, the facet's name is *tags*. Within the block, which is passed to the `facet` method, one has to specify the type of aggregation. In this example, the occurrence count of each tag shall be returned. Hence, the aggregation method is `terms`.

[35] http://www.elasticsearch.org/guide/reference/api/search/facets/
[36] http://karmi.github.io/tire/

The result object contains both the articles and the facet data by default. Figure 15 illustrates how the data can be accessed using the Tire result object. At first, a line is printed giving information on the number of articles matching the query and the articles' titles. Next, the facet result is presented in a tabular manner.

```
1 puts "Found #{s.results.count} articles:" +
2   " #{s.results.map(&:title).join(', ')}"
3
4 puts "Counts by tag:", "-"*25
5 s.results.facets['tags']['terms'].each do |f|
6   puts "#{f['term'].ljust(10)} #{f['count']}"
7 end
```

Listing 15: Result processing of the facet search example.

The facet data is stored in a `Hash` within the result set, using the facet name as key (see line five). Moreover, accessing the *tags* key also returns a `Hash` containing meta data for the facet which is not relevant for the example. The actual data can be reached with the key *terms*. This data is simply iterated over and printed. When running this script, the resulting output is:

```
Found 2 articles: Three, Two

Counts by tag:
-------------
ruby       1
python     1
java       1
```

Indeed, this is exactly what was expected. Only articles starting with a capital *T* are returned. Moreover, just the tags belonging to articles which match the query are counted.

For the sentiment tracking application it is not necessary to aggregate terms but to group tweets according to given time intervals. Fortunately, Elastic Search has a feature for this called **date histogram**[37]. Basically, a date histogram is just a facet able to handle dates. Listing 16 illustrates how to use a date histogram facet with Tire.

Filtering parameters, as shown in listing 8, are left out here. To retrieve a date histogram, the usual `facet` method is used, passing it the desired name, here *histogram*. Instead of calling `terms`, the method `date` is used now to initiate date interval grouping. Finally, one has to set the `field` containing the date and the grouping `interval`. Retrieving the data is very similar to the code presented in listing 15. For this reason, it is not shown again.

[37] `http://www.elasticsearch.org/guide/reference/api/search/facets/date-histogram-facet/`

```ruby
Tweet.search do
  # filtering left out here, see listing 8

  facet 'histogram' do
    date field: 'created_at', interval: interval
  end
end
```

Listing 16: Illustration of the date histogram usage in Tire.

4.7 Conclusions

Collecting and processing tweets from the Twitter Streaming API works fairly well using the Gems Tweetstream for the streaming and Resque for the asynchronous processing. Browsing data in tabular form is realizable easily by using Datatables. Presenting a sentiment time series can be done comfortably with Highcharts. Indexing, full-text searching and time series computation can be efficiently implemented using Elastic Search.

The application consists of 2,476 lines of code, the majority (1,660) being Javascript, and has been developed in slightly less than a man-month. Most of the Javascript code was necessary to connect and customize the client side components Datatables and Highcharts with each other and the backend. While there is not much Ruby code in relation to Javascript, it still took a while to figure out how to make the big variety of libraries and components work together. First and foremost, the largest part of the work was identifying the components which can be used to fulfill the requirements and integrate these components into the application. Due to their well designed interfaces not too much additional code was necessary.

In conclusion, developing a scalable real time sentiment tracking application can be done with relatively small effort by using publicly available tools only, once these tools have been identified and configured.

5 Reflexion

First of all, the results of the thesis are summarized and their transferability to other domains is discussed. Finally, an overall conclusion and an outlook with regard to further work in the thesis' direction is presented.

5.1 Summary and Transferability of Results

After motivating the general usefulness of Twitter Sentiment Analysis in chapter 1, chapter 2 introduced the basics necessary to understand the rest of the thesis. The standard machine learning based classifiers Naive Bayes and Support Vector Machine have been introduced. Next, it was presented how text can be transformed to vectors in various ways so that the classifiers can make use of them. Additionally, most of the current methods were presented and discussed.

In chapter 3, a comparison of the introduced methods, features and preprocessing techniques was performed. Due to the fact that most public data sets are suffering from certain drawbacks, quality criteria for a dataset were defined. Furthermore, a data set satisfying these quality criteria was created and analyzed. The analysis revealed that some kinds of features seem to be very good discriminators for the sentiment classes *positive* and *negative*, while others rarely occur in significantly different numbers in the two classes. After discussing how exactly the performance is measured, an experiment was conducted to determine the ideal size of the training corpus for each combination of classifier and feature. Next, the effects of preprocessing were investigated. The results of this investigation were fairly surprising. No preprocessing technique yielded significant improvements of the accuracy. That result is very astonishing since in current literature those techniques are applied with the claim that they would improve accuracy. However, they neither made the results worse while reducing the size of the feature space. Thus, performing the introduced preprocessing steps is not mandatory but can be used to reduce the memory consumption of the classifier. Finally, various combinations of features have been investigated for both introduced classifiers. No preprocessing was applied, and the corpus size, which was experimentally determined before, was used. Surprisingly, a combination of unigram and bigram features in conjunction with the Support Vector Machine performed best with an accuracy of 82.6%. Most authors of current literature claim that Naive Bayes Classifiers consistently outperform Support Vector Machines, which is obviously not true. It was also very surprising that while POS tags seemed to be features which separate the classes very well, a classifier trained with them performs significantly worse than the baseline. As a result, not all features which look promising at first sight are actually valuable for classification.

While this result mainly is relevant for the domain of Twitter Sentiment Analysis, it can be partly transferred to other domains. Other online communities may allow for longer

texts to be posted, but most of the time they are not much longer than a tweet. Hence, the classifier could probably also be applied to other online community messages.

The final chapter 4 presented an illustration on how a realtime sentiment tracking application can be designed and implemented as a web application. The major tools used for this were Ruby on Rails, Datatables, Highcharts and Elastic Search. Most of the work consisted of connecting these components to work together as desired. As a result, an application was created that is able to track the sentiment on the Twitter stream towards various entities, which are defined by a set of relevant keywords. The sentiment is presented as an intuitive chart with a variable resolution. Moreover, the user can browse the tweets in tabular form while filtering the results using a time frame and a full-text search.

Even though the application is centered on monitoring Twitter, it can be modified to be able to handle any online community with an API. Not too many adaptations would be necessary to implement that modification. Basically, just the data source has to be changed. Maybe one also would like to rename some classes like `Tweet` to `Message`. However, Twitter is the only community providing comfortable API access to its public stream right now. For this reason, such generalizations have not been implemented in this thesis.

5.2 Conclusion and Outlook

To conclude the performance investigation it is worth mentioning that current methods often discard other algorithms which actually perform on par with them. Moreover, various preprocessing techniques are claimed to increase accuracy. Since these claims are rarely backed with significance tests and experimental results, one should not simply believe them. The best method acquired in this thesis performs on par with most current methods even though it has been discarded by the majority of researchers. However, methods using hand crafted feature vectors outperform it significantly. As a result, handcrafted features are an interesting topic to be looked at in future work regarding Twitter Sentiment Analysis. One further interesting project could be the extension of the data set. It would be very interesting to see how classifiers trained only with hand labeled data perform. To be able to do this, a reasonable amount of hand labeled tweets needs to be available. Maybe so called crowd sourcing could be harnessed, which denotes the process of exposing an experiment publicly to the internet to acquire results. Since unknown people are less trustworthy in general, one would have to increase the number of validations. For example, only tweets could be taken into account where at least three labelers agreed on the sentiment.

The realtime tracking application provides an architectural example on which the implementation of such a platform for productive usage can be based. Once the tools have been brought in line, the actual coding effort is not very large. However, there is still room for improvements. For example, one could imagine various additional analysis steps being performed on the data. One possible step would be to analyze the content of the tweets automatically and present the user with a list of positive and negative aspects regarding the entities. The frequent occurrence of the phrase *customer support*

in negative tweets about a company, for example, may be a hint that the customer support is not working as intended. Another possible extension would be the automatic discovery of additional relevant keywords for the entity. The keywords given by the user could be treated as a seed set of keywords. Looking at the resulting tweets, the system may be able to determine which other keywords are relevant for the entity and could suggest these to the user.

All in all, this thesis provides a high quality data set for Twitter Sentiment Analysis, invalidates common prejudices regarding classifiers, features and preprocessing, and it finally illustrates an architecture for realtime sentiment tracking using the public twitter stream.

Bibliography

A. Agarwal, B. Xie, I. Vovsha, O. Rambow, and R. Passonneau. Sentiment analysis of twitter data. In *Proceedings of the Workshop on Languages in Social Media*, pages 30–38. Association for Computational Linguistics, 2011.

S. Asur and B. A. Huberman. Predicting the future with social media. In *Web Intelligence and Intelligent Agent Technology (WI-IAT), 2010 IEEE/WIC/ACM International Conference on*, volume 1, pages 492–499. IEEE, 2010.

A. Bakliwal, P. Arora, S. Madhappan, N. Kapre, M. Singh, and V. Varma. Mining sentiments from tweets. *Proceedings of the WASSA*, 12, 2012.

L. Barbosa and J. Feng. Robust sentiment detection on twitter from biased and noisy data. In *Proceedings of the 23rd International Conference on Computational Linguistics: Posters*, pages 36–44. Association for Computational Linguistics, 2010.

J. Bollen, H. Mao, and A. Pepe. Modeling public mood and emotion: Twitter sentiment and socio-economic phenomena. In *ICWSM*, 2011a.

J. Bollen, H. Mao, and X. Zeng. Twitter mood predicts the stock market. *Journal of Computational Science*, 2(1):1–8, 2011b.

B. Bonev, G. Ramírez-Sánchez, and S. O. Rojas. Opinum: statistical sentiment analysis for opinion classification. In *Proceedings of the 3rd Workshop in Computational Approaches to Subjectivity and Sentiment Analysis*, pages 29–37. Association for Computational Linguistics, 2012.

N. N. Bora. Summarizing public opinions in tweets. *Journal Proceedings of CICLing*, 2012.

B. E. Boser, I. M. Guyon, and V. N. Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 144–152. ACM, 1992.

C.-C. Chang and C.-J. Lin. Libsvm: a library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(3):27, 2011.

S. F. Chen and J. Goodman. An empirical study of smoothing techniques for language modeling. In *Proceedings of the 34th annual meeting on Association for Computational Linguistics*, pages 310–318. Association for Computational Linguistics, 1996.

C. Cortes and V. Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

N. A. Diakopoulos and D. A. Shamma. Characterizing debate performance via aggregated twitter sentiment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1195–1198. ACM, 2010.

R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. Liblinear: A library for large linear classification. *The Journal of Machine Learning Research*, 9:1871–1874, 2008.

K. Gimpel, N. Schneider, B. O'Connor, D. Das, D. Mills, J. Eisenstein, M. Heilman, D. Yogatama, J. Flanigan, and N. A. Smith. Part-of-speech tagging for twitter: Annotation, features, and experiments. Technical report, DTIC Document, 2010.

A. Go, R. Bhayani, and L. Huang. Twitter sentiment classification using distant supervision. *CS224N Project Report, Stanford*, pages 1–12, 2009.

P. E. Greenwood and M. S. Nikulin. *A guide to chi-squared testing*, volume 280. Wiley-Interscience, 1996.

N. Haldenwang. Agile Entwicklung einer Webapplikation zur Verwaltung und Auswertung von Klausur- und Übungsaufgaben in der Hochschullehre, 2011.

J. Han, M. Kamber, and J. Pei. *Data mining: concepts and techniques.* Morgan kaufmann, 2006.

B. J. Jansen, M. Zhang, K. Sobel, and A. Chowdury. Twitter power: Tweets as electronic word of mouth. *Journal of the American society for information science and technology*, 60(11):2169–2188, 2009.

F. Jelinek and R. Mercer. Interpolated estimation of markov source parameters from sparse data. *Pattern recognition in practice*, 1980.

T. Joachims. A probabilistic analysis of the rocchio algorithm with tfidf for text categorization. Technical report, DTIC Document, 1996.

T. Joachims. *Learning to classify text using support vector machines: methods, theory and algorithms.* Kluwer Academic Publishers, 2002.

K.-L. Liu, W.-J. Li, and M. Guo. Emoticon smoothed language models for twitter sentiment analysis. In *AAAI*, 2012.

D. J. MacKay and L. C. B. Peto. A hierarchical dirichlet language model. *Natural language engineering*, 1(3):289–308, 1995.

A. Mittal and A. Goel. Stock prediction using twitter sentiment analysis, 2012.

G. Neumann and S. Schmeier. Combining shallow text processing and machine learning in real world applications. In *Proceedings of the IJCAI-99 workshop on Machine Learning for Information Filtering, Stockholm, Sweden*, 1999.

O. Owoputi, B. O'Connor, C. Dyer, K. Gimpel, N. Schneider, and N. A. Smith. Improved part-of-speech tagging for online conversational text with word clusters. In *Proceedings of NAACL-HLT*, pages 380–390, 2013.

A. Pak and P. Paroubek. Twitter as a corpus for sentiment analysis and opinion mining. In *LREC*, 2010.

J. Read. Using emoticons to reduce dependency in machine learning techniques for sentiment classification. In *Proceedings of the ACL Student Research Workshop*, pages 43–48. Association for Computational Linguistics, 2005.

D. Roth and D. Zelenko. Part of speech tagging using a network of linear separators. In *Coling-Acl, The 17th International Conference on Computational Linguistics*, pages 1136–1142, 1998. URL `http://cogcomp.cs.illinois.edu/papers/pos.pdf`.

H. Saif, Y. He, and H. Alani. Alleviating data sparsity for twitter sentiment analysis. In *The 2nd Workshop on Making Sense of Microposts*, 2012a.

H. Saif, Y. He, and H. Alani. Semantic sentiment analysis of twitter. In *The Semantic Web–ISWC 2012*, pages 508–524. Springer, 2012b.

H. Schmid. Probabilistic part-of-speech tagging using decision trees. In *Proceedings of international conference on new methods in language processing*, volume 12, pages 44–49. Manchester, UK, 1994.

H. Schmid. Improvements in part-of-speech tagging with an application to german. In *In Proceedings of the ACL SIGDAT-Workshop*. Citeseer, 1995.

V. Vapnik. Estimation of dependencies based on empirical data, translated by s. kotz, 1982.

V. Vapnik. *The nature of statistical learning theory*. springer, 2000.

V. N. Vapnik. Statistical learning theory. 1998.

J. C. Ward and A. L. Ostrom. The internet as information minefield: an analysis of the source and content of brand information yielded by net searches. *Journal of Business research*, 56(11):907–914, 2003.

E. Yoon, H. J. Guffey, and V. Kijewski. The effects of information and company reputation on intentions to buy a business service. *Journal of Business Research*, 27 (3):215–228, 1993.

Q. Yuan, G. Cong, and N. M. Thalmann. Enhancing naive bayes with various smoothing methods for short text classification. In *Proceedings of the 21st international conference companion on World Wide Web*, pages 645–646. ACM, 2012.

C. Zhai and J. Lafferty. A study of smoothing methods for language models applied to information retrieval. *ACM Transactions on Information Systems (TOIS)*, 22(2): 179–214, 2004.

Erklärung

Ich versichere, dass ich die eingereichte Master-Arbeit selbstständig und ohne unerlaubte Hilfe verfasst habe. Anderer als der von mir angegebenen Hilfsmittel und Schriften habe ich mich nicht bedient. Alle wörtlich oder sinngemäß den Schriften anderer Autoren entnommenen Stellen habe ich kenntlich gemacht.

Osnabrück, den 24.09.2013

(Nils Haldenwang)