FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Radovan Jankovič

Artificial neural networks and their application in text analysis

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis: doc. RNDr. Iveta Mrázová, CSc.

Study programme: Software Systems

Specialization: Database Systems

Prague 2016

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague 20.7.2016

I wish to thank the supervisor for numerous valuable remarks and corrections and their overall guidance that shaped this thesis.

Title: Artificial neural networks and their application in text analysis

Author: First and last name of the author

Department / Institute: Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis: doc. RNDr. Iveta Mrázová, CSc., Department of Theoretical Computer Science and Mathematical Logic

Abstract: This thesis is devoted to the area of sentiment analysis. Its goal is to discuss and compare various methods applicable to sentiment classification of short texts. When analysing the described techniques, we will orient ourselves towards the context of social networks. Recently, this type of media became the source of vast amounts of data and the demand for its automatic processing is high. Interesting results have been obtained for clustering used in combination with supervised learning and convolution, which is primarily used for image data.

Keywords: machine learning, sentiment analysis, clustering, social networks

Contents

Introduction

Social networks, like Facebook or Twitter, are a phenomenon that has recently transformed numerous aspects of our lives. The impact of these platforms is no longer limited to entertainment or personal presentation of individuals. Indeed, they have formed a new sphere of business, with some emerging companies focusing primarily on this platform and others, more traditional ones, expanding their marketing efforts there.

The companies that seek to present themselves on social networks understand that new methods are necessary to communicate with their customers efficiently and to manage their virtual identity properly. Among other factors, this is caused by the enormous amounts of data users can provide, its varying quality and form. On the other hand, the feedback a company receives through this new channel is very compact, direct and informal in its nature.

Intense research is being performed in order to efficiently categorize, filter and interpret the customer-generated content on social networks. One of the key segments of this effort is sentiment analysis. Its goal is to interpret the sentiment of the posts, comments or other forms of reactions users generate in relation to a company or a product. The ability to accurately estimate the opinion of potentially millions of users involved with a product or service, and to pinpoint the key aspects of the product on which this opinion is based, is of an unquestioned value.

This research is oriented towards the basic task of text message classification on a sentiment scale from negative to positive. The text compares various methods that can be used for sentiment classification, with multiple chapters representing a step towards the design of a modified method based on the observations made in the previous one.

The first chapter of the thesis provides an overview of data mining, briefly describing the phases of a datamining process. It yields an overall picture before we focus on the learning algorithms, whose application typically represents only a fraction of the overall workload.

Chapter 2 discusses logistic regression, being a relatively simple method that manages to provide reasonable and well-interpretable results. Chapter 3 takes a more complex approach, namely the syntactic analysis of the text, using recursive neural tensor networks.

Chapter 4 simplifies the entire approach by analyzing the support vector machine models. These models render reasonable results and the following chapters attempt to further improve on the performance of SVMs. Chapter 5 focuses on slang terms and phrasal expressions, which are especially frequent on social networks, attempting to appropriately translate them so that they can be interpreted by a classifier properly.

Chapter 6 takes a sidestep by exploring self-organizing maps. It was discovered that the clusters formed by these maps are in some instances correlated to sentiment. This occurred in spite of the training process being unaware of the sentiment class. SOM models were combined with SVMs and a proposed hybrid model was tested.

Chapter 7 deals with the technique of convolution, often used to reduce the computational difficulty of training neural networks for image recognition. Here the text explains how to apply it to text classification. Chapter 8 describes an application for testing of selected models discussed through this work, namely SVMs, convolutional networks and their combination with SOMs, as discussed in chapters 4, 6 and 7. It also gives a more detailed description of the preprocessing typically applied before training the models.

1. Overview of data mining

The need to store and manage information has been present for thousands of years. Knowledge has proven to be a decisive factor in countless cases though history and so Robin Morgan did little in the terms of discovery when he stated that knowledge was power.

The last few decades have seen immense advances in the means of collection, storage, transmission and processing of data. The global data storage capacity has risen from 2.6 exabytes to 15.8 in 1993 and 295 in 2007 [1]. This growth is exponential not only in the theoretical capacity, but more importantly in the actual amount of data stored, which was estimated to double every 20 months [2]. This phenomenon has been referred to as the digital revolution or the information age. If we accept information as a valuable asset and consider its high availability, one could assume that the information age will be a golden age, when for each and every decision all related aspects can be taken into consideration and, using this information, the correct decision can be devised. This assumption would, however, clearly not correspond to reality. On the contrary, information overload, the state when one is presented with information which is difficult to understand not necessarily due to the complexity of its items, but rather by the sheer volume, has been noted in several fields and is easily observable in everyday life.

This situation has prompted the development of a new field in information technology, one which can be defined in multiple ways and perhaps is better characterized by its aims than by the processes and methods it employs. This field is known as data mining and the aim is to convert data to useful information. In other words, raw facts, often in vast quantities, are to be transformed into organized, presentable and meaningful information that people can reason about effectively. By doing so, one of the key difficulties brought by the information age can be addressed. The following sections provide an overview of the typical premises and stages of a data mining process.

1.1. Data mining stages

The obvious requirement of data mining is the presence of data. This data originates from deployed information systems or, in general, data-related business processes and the aim of data mining is to aid these processes. Therefore, task-specific data mining projects are usually not developed as a basis for other systems, but rather as an extension of a process already in place. There are, of course, exceptions in the form of general-purpose data mining frameworks and algorithms, but in use these practices, adjustments are again necessary in order to match the characteristics of the problem at hand.

This makes describing a data mining process in general rather difficult. However, there are basic patterns that usually accompany such a process. Several descriptions of these patterns have been formulated in recent years. One of the most popular is the Cross-Industry Standard Process for Data Mining, shortly CRISP-DM [3], which describes a six-stage data mining process.

1.1.1. Business understanding

This is a common first step for many software systems. The task has to be clearly defined by the business users and a preliminary analysis is needed so that the principal flaws are discovered. After this phase, the general aim should be known, alongside with the basic concepts of possible solutions.

1.1.2. Data understanding

During this stage, the available datasets should be briefly examined to determine the sources that will be built upon. Data quality, the level of detail and volume of the data should be considered. All participants should be familiar with these datasets on a basic level in order to work with them effectively in later stages.

1.1.3. Data preparation

Before data mining techniques can be applied, suitable data subsets need to be selected. These often combine data from multiple sources, in which case the datasets must be integrated and data consistency must be ensured. Special cases, such as missing values or outliers, also have to be taken care of. Additional calculations and transformations are often necessary to create datasets that are simple in structure, yet comprehensive and well-suited for further processing.

Clear and consistent data is a valuable asset in itself and is the aim of many data modelling projects, giving the analysts a viable tool to work with. Further steps focus on the automation of data analysis.

1.1.4. Modeling

In this phase, various data mining, or rather machine learning algorithms are employed and their performance on the required tasks is measured. These algorithms perform the essence of data mining – the transformation of data into some sort of information. The best algorithms are selected and their parameters are calibrated for further optimization. Specific algorithms often have additional requirements on the data and so the data preparation phase has to be revisited to make enhancements to the datasets.

Information has been also gained in the previous step, but there is a significant difference. Data preparation cleans and structures the data in a way that inner relations can be observed and analyzed more easily, but does not create new information directly. New insights have to be delivered by analysts. Machine learning algorithms can often formulate hypotheses and predictions based on the data and present it to the user directly.

Many works in the field of data mining focus solely on machine learning algorithms in terms of their performance. This is understandable as these algorithms are interesting mathematically and reusable for an array of similar tasks. While we will emphasize the algorithms as well, we will apply these to real-life data originating from the web, where the preparation phases are necessary. A broader introduction is therefore needed. Also, while precision of machine learning algorithms is a suitable comparative measure, interesting insights into the data that cannot be expressed in figures can occur and are, from a practical point of view, desirable.

1.1.5. Evaluation

After the modelling phase, the performance of the resulting models is evaluated. The whole data mining process so far should also be reviewed as execution of each step affects the results of all following steps. Finally, the decision has to be made, whether the proposed models fulfill the business requirements and how the deployment of the solution should be performed.

1.1.6. Deployment

In the final step, the model has to be deployed to the company environment. This has to be done in cooperation with the customer so that the new system is suitably incorporated in the company`s infrastructure and business processes. Users also have to be trained to use the system effectively and shown its benefits to use it at all. The system, on the other hand, must be made easily accessible and intuitive, if possible.

1.2. Data mining models

The result of data mining should be a useful set of information. What form this information is supposed to have, is dictated by the specific task and business requirements. Still, these forms can be divided into few a basic classes.

1.2.1. Description and discrimination

Descriptive models alleviate the comparison of known classes of objects. To make this comparison easier, different sorts of data is gathered and made available in a comprehensive form that allows the browsing of different attributes of each class. This can be realized by the simple means of a set of pivot tables and charts, databases where additional statistics can be computed easily or complex multidimensional cubes and OLAP analysis using these cubes.

For example, a company with many salesmen can distinguish between the more and less successful ones based on their return on investment, but cannot clearly describe what led to these differences. With the help of a multidimensional cube, these groups can be compared in terms of their experience, number of customer visits and other activities they perform. Generally, their working habits can be described and set as an example for others.

1.2.2. Classification and numeric prediction

Classification models are able to, given a description of an entity, predict the class into which this item belongs. The classes are defined in advance and the model learns based on training examples to which the correct labels have been assigned in advance, usually manually. Apparently, these models will internally seek a description or discrimination of the classes, as the users would in the previous case. However, the characterization the algorithms produce is formulated in a way that can be then used for automatic classification. These models can have a wide variety of forms, from the more comprehensive, such as sets of “if-then” rules to rather enigmatic ones, such as artificial neural networks.

Numeric prediction, often also called regression due to regression being the most popular method, is similar to classification with the main difference that it predicts a value on continuous scale rather than a class from a finite set.

Many predictions that have been investigated by statisticians and commercial businesses alike, fall to this category of models, such as illness or malfunction diagnostics, financial forecasts and text or multimedia classification.

1.2.3. Clustering

Clustering algorithms are similar to the previously mentioned models in the sense that they also divide the data into certain classes. These classes are, however, not known in advance and therefore, obviously, the correct classification of instances is also unknown. This is, arguably, a rather benevolent set of requirements for an algorithm and it is often also difficult to decide which model is better, since we do not have clearly defined performance measures [4]. With clustering, the aim is to form clusters such that the instances within each cluster are similar and at the same time each two clusters differ.

1.2.4. Association

Associative algorithms, as the name suggests, form associations between attribute values based on their occurrence in the data. These can be items that frequently appear together in a market basket in a grocery or groups of films viewed. Based on these associations, a set of if-then rules can be formed. These are then scored based on how often they appear in the data and what is the probability of the conclusion of the correctness of the rule when the premise is fulfilled.

1.3. Machine learning and data

Machine learning investigates how computers can learn (or improve their performance) based on data. A main research area seeks to enable computer programs to automatically learn to recognize complex patterns and make intelligent decisions based on data [2].

The data the algorithms learn from is usually a dataset of instances, each instance characterizing a single object. Multiple aspects of the objects can be described in the instance. We will call these aspects attributes and denote the instance as a vector of its attributes: $\boldsymbol{x} = (x_1, x_2, \dots, x_n)$. The attributes will in most cases be numeric values, a description of the domain of each attribute will be given otherwise. Sets of instances will also be referred to as datasets. Available data will often be divided into train, validation and test datasets. The train dataset is the dataset the algorithm learns from, builds its model based on it. The validation dataset (if present) is used to tune the parameters of the algorithm. The test dataset is used to measure the performance of the final model.

2. Logistic regression

Logistic regression is a method for modeling the dependency of a given variable y on a vector of independent variables $\boldsymbol{x} = (x_1, x_2, \dots, x_n)$, $\forall i: x_i \in \mathbb{R}$. Both the dependent and the independent variables usually represent certain attributes of a given entity, such as different measures related to the performance and equipment of a car, the weather conditions, etc. In this section, we will use logistic regression for a simple sentiment analysis, more specifically for classifying texts as being of positive or negative sentiment.

The assumption is that y is categorical. In the basic, binary form of logistic regression, y can only have one of two values, 0 and 1. In this respect, the logistic regression is based upon linear regression, to which it bears many similarities. We will therefore begin the discussion of logistic regression with a brief look at the method of linear regression.

Given a vector of independent variables $\boldsymbol{x} = (x_1, x_2, \dots, x_n)$**,** the linear regression model produces an estimate of the dependent variable y, $\tilde{y}$. The estimate is computed as a linear combination of the independent variables

$$\tilde{y} \;\; = \boldsymbol{\beta x} + \beta_0 = \sum_{i=1}^{n} \beta_i x_i + \beta_0 \qquad (1)$$

where $\boldsymbol{\beta} = (\beta_1, \beta_2, \dots, \beta_n)$ and β_0 are parameters to be learned. While linear regression is easy to interpret and can work well for numeric predictions, it is not ideal for classification tasks.

Given the binary classification problem, we would often like to interpret the estimate $\tilde{y}$ as the probability of the object at hand belonging to a certain class or not. Based on this, we expect the estimate to fall into the interval $[0,1]$. This, however, is not guaranteed by the linear regression model. To address this issue, logistic regression introduces a transformation of the linear regression's result. While with linear regression the linear combination of the members of the independent variable vector is used directly as the output of the model, logistic regression uses the logit function to transform the potential interval of values of the linear combination, $(-\infty, \infty)$, to a probability ranging from 0 to 1. The logit function is defined as

$$\sigma(t) = \frac{1}{1 + e^{-t}} \qquad (2)$$

When applied to the linear regression model, it gives us the following expression for the estimate

$$\tilde{y} = \frac{1}{1 + e^{-(\beta_0 + \boldsymbol{x\beta})}} \qquad (3)$$

The inverse of the logit function is

$$\sigma^{-1}(t) = \ln\left(\frac{t}{1-t}\right) \tag{4}$$

which means that the linear combination of independent variables now corresponds to

$$\boldsymbol{\beta x} + \beta_0 = \sigma^{-1}(\tilde{y}) = \ln\left(\frac{\tilde{y}}{1-\tilde{y}}\right) \tag{5}$$

2.1.Limitations

Given a sample item, the label estimated by a logistic classifier is 1 when $\tilde{y} > \frac{1}{2}$ and 0 when $\tilde{y} < \frac{1}{2}$ or, in terms of the learned attribute weights $(\boldsymbol{\beta}, \beta_o)$, the estimated class is 1 when $\beta_0 + \boldsymbol{x\beta} > 0$. However, thanks to the adjustments made, we can treat $\tilde{y}$ as a proper probability and keep its distance from $\frac{1}{2}$, the "indecisive" boundary, as a measure of certainity of the classifier.

The logit function also makes logistic regression capable of handling nonlinear dependencies. On the other hand, this ability is limited – for example, the binary function XOR cannot be learned properly. Generally, one attribute cannot impose negation on the value of other attributes, as they appear in a linear combination and the model assumes their independence. While fairly few would consider building a classifier for XOR, this limitation can be critical when logistic regression is used for text classification, where it cannot cope with negation even when clearly stated. All it can do is to lean to a certain class when words such as "no" or "not" are encountered.

In this respect, the classifier has basically the same capabilities as a single perceptron.

The model can also be extended for multinomial classification. For c classes $0, 1, \dots, c-1$, this requires a separate set of parameters $(\boldsymbol{\beta}^{(c)}, \beta_0^{(c)})$ for each class. We can compute the probability of the object belonging to each class in turn:

$$P(Y = c|X = x) = \frac{e^{\beta_0^c + \boldsymbol{\beta}^c x}}{\sum_{k=0}^{c-1} e^{\beta_0^k + \beta^k x}} \tag{6}$$

This corresponds to the binomial case, which can also be written as

$$\tilde{y} = \frac{e^{\beta_0 + \boldsymbol{\beta} x}}{1 + e^{\beta_0 + \boldsymbol{\beta} x}} \tag{7}$$

It has also been argued that logistic regression in most cases does not preform significantly better than linear version, while the linear model is easier to interpret [5]. The problem with results outside the interval [0,1] could be effectively solved by substituting the closest of the two boundary values and tests have shown correlation between the results of logistic and linear regression close to 1. However, logistic regression was not outperformed by linear regression. Logistic regression does not have the flaws of linear mentioned previously, although minor, and the mechanics of the logistic method makes it easily comparable with more complex methods, specifically artificial neural networks, as we will see later.

2.2.Optimization

Before we look at the optimization of logistic models, we will shortly introduce the Newton's general method for optimizing differentiable functions, which will be applied later in this section.

2.2.1. Newton's method

Newton's optimization method, given a differentiable function f and a starting point x_0, constructs a sequence $x_1, x_2, \dots$ that converges to a point x where $f(x) = 0$. In eah iteration, the method takes a step from the prievious position x_i by finding the tanget of f at x_i and the point where this tangent crosses the x-axis. This is the new approximation, x_{i+1}:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{8}$$

Newton`s method in most cases converges faster than gradient methods, but can become costlier when used in its generalised form for functions of multiple variables.

2.2.2. Optimizing logistic regression

Since logistic regression predicts probabilities, we can use the likelihood measure for its optimization. Given n instances of the independent variable vector $\boldsymbol{x}_1, \boldsymbol{x}_2, \dots, \boldsymbol{x}_n$, expected results $y_1, \dots, y_n, \forall i: y_i \in \{0,1\}$ and estimates $\widetilde{y_1}, \widetilde{y_2}, \dots \widetilde{y_n}$, we compute the likelihood as a function of the learned parameters

$$L(\boldsymbol{\beta}, \beta_0) = \prod_{i=1}^{n} \tilde{y}_i^{y_i} (1 - \tilde{y_i})^{1-y_i} \tag{9}$$

To simplify and better decompose the expression to be optimized, we can apply logarithm to the right hand side, which gives us the log-likelihood function that we will denote l. We will also use the symbol $\boldsymbol{\theta}$ to refer to the vector of all the learned parameters, $(\boldsymbol{\beta}, \beta_0)$.

$$l(\boldsymbol{\theta}) = l(\boldsymbol{\beta}, \beta_0) = \log(L(\boldsymbol{\beta}, \beta_0)) \tag{10}$$

The likelihood is convex, thus finding the optimum means finding a set of parameters for which the derivative $l'(\boldsymbol{\theta}) = 0$. One popular option is using the Newton`s method, which, when applied to the derivate of the log-likelihood logistic regression, is also called Fisher scoring.

The formula for the next optimization step will be

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \frac{l'(\boldsymbol{\theta}_i)}{l''(\boldsymbol{\theta}_i)} \tag{11}$$

2.3. Logistic regression applied to text mining

In the following experiments we will be using the Sentiment140 Stanford training dataset [6]. This dataset contains 1.6 million tweets with sentiment annotations for 2 classes – positive and negative, $800\ 000$ instances for each class. To simplify the tests, we will only consider instances in the positive and negative classes. We will also reduce the dataset to $11\ \ 722$ randomly chosen instances to speed up the training and make more experiments feasible. In this reduced set, positive tweets slightly outnumber the negative ones – $6\ 156$ to $5\ 556$. We will be using the „Logistic“ method implementation from Weka 3.6 [7], which builds a multinomial logistic regression model.

The aim is to find the parameters that influence the performance of the classifier the most. The models were tested using 10-fold cross validation.

2.3.1. Default setup

The default data mining procedure used was as follows.

- A dictionary of expressions occurring in the dataset was formed. The expressions were $1-5$ word long (ngrams). The tweets were converted to vectors of $0/1$ attributes, one for each of the expressions in the dictionary. The value is 1 if the expression occurs in the tweet, 0 otherwise.
- From these attributes, the top 1000 were chosen, based on the information gain[1] of each attribute.
- The logistic regression model was learned in each stage of the 10 -fold crossvalidation.

2.3.2. Number of attributes and stopwords

The number of attributes (the top n 1-5-gram expressions used as attributes of instances) showed to have the largest impact on the precision, as shown in Table 1 below.

[1] Information gain is defined the difference between the entropy of a random variable H when certain information a is not known and when this information is known: $I(X,a) = H(X) - H(X|a)$. For a random variable X , $P(X)$ its mass probability function, entropy is defined as $E[-\ln(P(X))]$.

A	10	50	100	200
P	0.6204	0.6737	0.6965	0.7030
A	400	600	800	1000
P	0.7395	0.7541	0.7645	0.7722
A	1500	2000	3000	
P	0.7842	0.7937	0.8069	

TABLE 1: *THE EFFECT OF THE NUMBER OF ATTRIBUTES USED ON THE PRECISION OF LOGISTIC REGRESSION WHERE THE ATTRIBUTES ARE INDICATORS OF 5-GRAMS. A STANDS FOR NUMBER OF ATTRIBUTES, P FOR PRECISION.*

Improved performance as a result of the expansion of the dictionary is understandable, as there are many positive and negative expressions in the English language which may not have been covered previously. Relatively good results were achieved considering the model does not take into account the order in which the expressions occur in the tweet and does not perform any syntactic decomposition of the sentences. What could be surprising is the relative good precision with as few as 10 attributes.

Information gain	Attribute
0.01269	sad
0.01179	thanks
0.00794	miss
0.00694	sick
0.00652	sorry
0.00642	hate

TABLE 2: *ATTRIBUTES (EXCLUDING STOPWORDS) WITH THE HIGHEST INFORMATION GAIN*

The surprising finding though was that the word „i“ had the highest information gain over all, indicating positive sentiment. This is in accordance with a recent study of the effects of social networks [8]. While social networks theoretically allow for truthful representation of one’s life, most people, being the editors of their own virtual profiles, naturally tend to present themselves positively, focusing on their achievements and well-being rather than their problems when presenting themselves to the society. Similarly, the 5th most positive word was „you“. So we could say that the positive bias of social networks does not affect only the information one publishes about themselves, but also direct interactions with other people. This could be explained as politeness or the opposite, talking behind one`s back. Such theories are beyond the scope of this section, though.

On the other hand, it could be argued that words such as „i“ and „you“ should not be included in the attributes as they are stopwords. When stopwords were filtered out from the attributes, the resulting precision was about 1% lower than in the previous case. Therefore, the exclusion of stopwords does not harm the results significantly and generally should be performed although in the case of social networks certain stopwords can indicate the sentiment surprisingly well, as shown above.

2.3.3. The use of 5-grams

By default, 5 -grams were used because it was expected that multi-word expressions would capture the sentiment better than separated words. To test this, the previous experiment was repeated with single words or 1-grams. The results are shown below.

A	10	50	100	200
P	0.6204	0.6838	0.7073	0.7396
A	400	600	800	1000
P	0.7614	0.7746	0.7897	0.7829
A	2000	3000	4000	
P	0.7583	0.7233	0.7123	

TABLE 3: ***THE EFFECT OF THE NUMBER OF ATTRIBUTES USED ON THE PRECISION OF LOGISTIC REGRESSION WHERE THE ATTRIBUTES ARE INDICATORS OF OCCURRENCE SINGLE WORDS. A STANDS FOR NUMBER OF ATTRIBUTES, P FOR PRECISION.***

Although the model shows good results with fewer attributes, which are even slightly better than the 5-gram alternative, but the performance diminishes when there are more than 1000 attributes. This is probably caused by overfitting, to which logistic regression is vulnerable. There are obviously more $1-5$ grams whose sentiment can be clearly determined than 1-grams. Probably, when the model is allowed to use too long phrases, it takes advantages of attributes that do not really indicate sentiment, but rather randomly correlate with the confidence of the test data. These correlations then do not apply well to test data.

2.3.4. Ridge parameter

Often, the attributes chosen for logistic regression are highly correlated and thus the regression parameters do not represent any inherent effect of a given attribute, but depend on other attributes included [9]. To aid this we can place a small penalty on the norm of $\boldsymbol{\beta}$. To do this, we add the square of the norm of $\boldsymbol{\beta}$ to the log-likelihood which is to be maximized.

$$l(\boldsymbol{\theta}) - \lambda\|\boldsymbol{\beta}\|^2 = \log\big(L(\boldsymbol{\beta}, \beta_0)\big) - \lambda\left(\sqrt{\sum_{i=1}^{n} \beta_i^2}\right)^2 \qquad (12)$$

Here λ is the ridge parameter. In the experiments described above, 1.10^{-8} was used as the default value of the ridge parameter. A few other values were tested, but did not lead to a significant change in the precision of the model. In fact, the model learned better when there was no ridge present, which indicates that the there was no significant collinearity between the attributes.

Ridge	0	1.10^{-8}	1.10^{-4}
Precision	0.739976	0.739891	0.739891

TABLE 4: **THE** ***EFFECT OF VARIOUS VALUES OF THE RIDGE PARAMETER.***

2.3.5. Less significant parameters

Other methods of ranking 5-grams to choose the top 1000 attributes were also tested; the results were almost identical when either correlation coefficients or principal component transformation was used.

3. Recursive neural tensor networks

While the logistic regression method is simple and performs quite well, it has apparent shortcomings. One of the main disadvantages of this and similar methods representing the text as a bag of words, basing the models only on the word's occurrence, is that the structure of the sentence is disregarded. This makes it very difficult for the model to cope with negation and conditional sentences. For longer texts, these methods can achieve high accuracy when focusing only on strong, decisive sentiment indicators, however for short texts such as tweets it has been shown that these methods generally do not exceed 80% accuracy [10].

Two main approaches can be used to remedy this. The more widely used one is based on unsupervised learning and a lexicon of phrases and structures. The lexicon is collected manually, usually for applications in a specific field, as the knowledge of common expression patterns often does not transfer to other domains well [11]. Although the lexicon-based systems can achieve a good accuracy and are the primary approach taken on the industrial level, the creation of lexicons is laborious and requires good knowledge of the field.

The other option is to rely on supervised learning to capture the structure of a sentence. These methods often begin with parsing the sentences of the text and possibly detecting some structural patterns, followed by annotating the parse tree with low-level (word or letter) features. Finally, an aggregate function is applied to produce the resulting sentiment score.

If we decide to use supervised learning, the basic problem becomes the composition of phrases. In order to be able to handle phrases of any length and structure, from single words to complex sentences, using one method, this method needs to compose words into phrases and parts of a phrase into one. Ideally, we want this composition to copy the natural process of forming a sentence or a longer text, where each segment has a meaning and possibly sentiment of its own. If we follow the hierarchical structure of a sentence bottom-up, combining its segment one step at a time, we can also check the process, not only the result, of the algorithm and reason about each step of the composition. This is a valuable asset in machine learning, where the computation process is often enigmatic and evaluated purely on the precision of the final result. Figure 1 demonstrates the benefits of this approach on a semantic level.

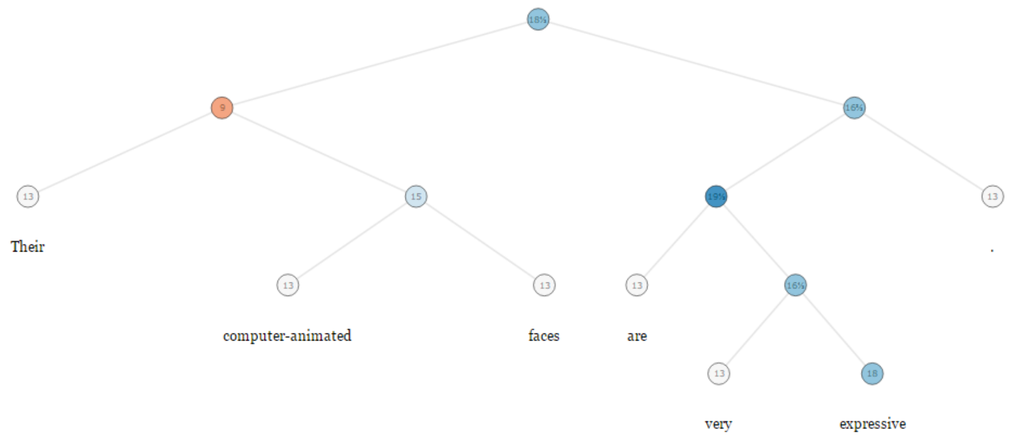

Figure 1: A composition tree of a simple sentence. The tree correctly identifies the word "expressive" as bearing a positive sentiment while recognizing that "computer-animated" and "faces" are neutral words, but the expression "their computer-generated faces" is rather negative, giving an inhumane impression. That is, until it is combined with the positive second part of the sentence that shows the author's appreciation of these faces.

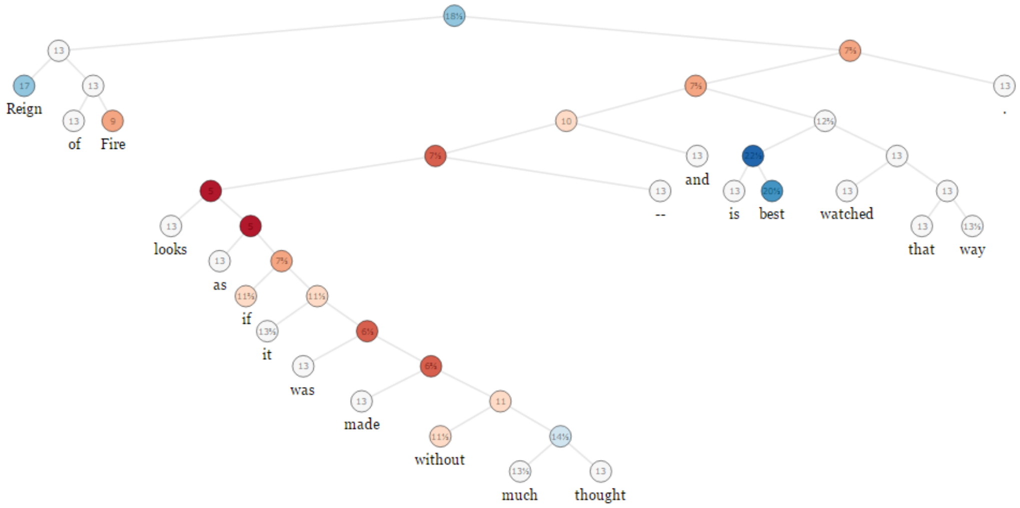

Figure 2: A more questionable composition tree - we can see correct behaviour in the negative subtree on the left, where the negative sentiment builds up as the phrase is developed. Both children of the root node are probably evaluated correctly. However, is the result truly positive? Similar cases occurred also in our experiments – see 3.2.

In order to use recursive neural networks, we need, however, training data that provide annotation not only on the sentence or document level, but also for each phrase on each level of the parse tree. Such a dataset is now publicly available from the Stanford Sentiment Treebank, which consists of over 200 000 phrases parsed from 11 855 reviews. Each of these reviews has been annotated by three human judges on a 25-degree scale (although in most applications only the 5 main levels are used – very negative, somewhat negative, neutral, somewhat positive, very positive).

3.1.The computational model

We will now explain the principles of recursive neural tensor networks. These were proposed by [10] as an improvement to previous work on recursive neural networks, which will be discussed first. Both these methods operate on the parse tree produced by the Stanford parser [12], which parses a sentence into a binary tree based on part-of-speech tags (POS tags). Each node, which, at the leaf level, is a word, is represented by a d-dimensional vector. The leaf level vectors are randomly initialized and the matrix $\boldsymbol{L} \in \mathbb{R}^{d \times |V|}$ where $|V|$ is the size of the vocabulary. $\boldsymbol{L}$ is a parameter to be trained.

Also common to the models is the function used to compute the final sentiment from the d-dimensional node vector $\boldsymbol{a}$. The sentiment is given by

$$y_a = softmax(\boldsymbol{T}_s \boldsymbol{a}) \tag{13}$$

where $\boldsymbol{T} \in \mathbb{R}^{c \times d}$ is the sentiment classification matrix to be trained and c is the number of classes we want our classifier to use. In Socher's demonstration, $c = 5$, which is sufficient, since five classes were discovered to cover the majority of the choices made by human annotators even when given the option to use 25 degrees of sentiment.

The difference between the two models is the composition function, which is the method by which two node feature vectors are combined into one. Generally, we construct an aggregate function $u: \mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R}^d$.

In the older model of recursive neural networks as described by [13], the composition is defined as

$$f(\boldsymbol{a}, \boldsymbol{b}) = g\left(\boldsymbol{W} \begin{bmatrix} \boldsymbol{a} \\ \boldsymbol{b} \end{bmatrix}\right) \tag{14}$$

where $\boldsymbol{W} \in \mathbb{R}^{d \times 2d}$ is the main parameter to be learned and g is $tanh$ applied element-wise.

Tensor neural networks extend this model by allowing for more complex interactions between the input vectors. This is achieved via a tensor of bilinear forms $\boldsymbol{V} \in \mathbb{R}^{d \times 2d \times d}$. The tensor product $h \in \mathbb{R}^d$ is given by

$$h = \begin{bmatrix} a \\ b \end{bmatrix}^T \boldsymbol{V}^{[1:d]} \begin{bmatrix} a \\ b \end{bmatrix}; h_i = \begin{bmatrix} a \\ b \end{bmatrix}^T \boldsymbol{V}^i \begin{bmatrix} a \\ b \end{bmatrix} \tag{15}$$

The overall composition of the two vectors is

$$p = g\left(\begin{bmatrix} a \\ b \end{bmatrix}^T \boldsymbol{V}^{[1:d]} \begin{bmatrix} a \\ b \end{bmatrix} + \boldsymbol{W} \begin{bmatrix} a \\ b \end{bmatrix}\right) \tag{16}$$

with the $\boldsymbol{V}$ and $\boldsymbol{W}$ parameters to be optimized. Both $\boldsymbol{V}$ and $\boldsymbol{W}$ can be extended by a $\mathbf{1}$ column to account for bias, while extending the concatenation of the node vectors accordingly. Figure 3 illustrates this function in the 2-dimensional case.

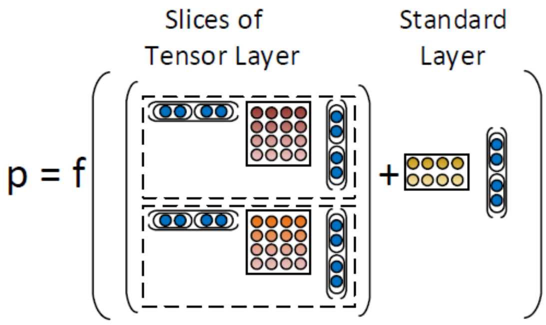

FIGURE 3: ONE LAYER (NODE IN THE PARSE TREE) OF A RECURSIVE NEURAL TENSOR NETWORK FOR D=2. THE V MATRIX IS DISPLAYED IN RED, W IN YELLOW.

3.2. Experimental results

Experiments performed by the authors showed that recursive neural tensor networks (RNTN) outperform the basic recursive neural networks, while both models achieved maximum performance for vector sizes between 20 and 30. When the vectors were longer or shorter, accuracy dropped. This indicates that tensor networks are more successful not simply due to having more parameters, but thanks to being able to better interpret complex sentences. This was proven by analysing two phenomena in particular – negations and contrastive sentences („X but Y"). In both cases, the networks performed well. Overall accuracy over 85% was reported for 5-class labeling of the entries of Stanford Sentiment Treebank and over 80% for similiar classification on the phrase and word levels.

3.3. Twitter sentiment evaluation experiments

The neural tensor network sentiment classifier is available from the Stanford University NLP portal (http://nlp.stanford.edu:8080/sentiment/), alongside with the rest of the NLP framework. To test the system, a manually annotated dataset was used. This dataset originates from subtask A of task 4 of the annual SemEval competition [14]. This competition covers a wide area of NLP-related topics including taxonomy, textual similarity and sentiment analysis. The dataset consists of 3915 tweets annotated as being positive, negative or neutral, with approximately even distribution.

Although the classifier was already trained using the movie review dataset described before and ready to use, additional processing was necessary to account for the specifics of tweets. Most notably, there were three aspects. Firstly, the sentiment evaluator works on the sentence level while twitter allows only for messages 140 characters long and thus many tweets are indeed single sentences, the majority consists of multiple. An aggregation function was needed to compose the sentence-level sentiments.

Secondly, tweets are more informal, use slang, emoticons and twitter-specific constructs such as #-annotations and @-references. One possible approach was to simply exclude all these noisy elements. But that would mean disposing of a large part of the information in the tweets. Words beginning with „#" are tags that the user chose and are therefore a form of keywords rather than noise. Similarly, emoticons, though they may not be proper written expressions in most aspects, pose a significant sentiment indicator.

Finally, the classifier had a negative bias. A small positive bias was initially added to counter this, but this approach is clearly not ideal. Using this “counterbias” and various modifications to the text, such as recognizing emoticons and adjusting the sentiment score according to their presence, results with precision ranging from 60% to 80% were obtained. However, in later experiments we abandoned the compensation bias and focused more on the cause of the negative bias.

It was later discovered that multiple users faced the problem with negative bias according to the discussion under the page hosting a live demo of the sentiment classification engine [15]. For example, the clearly positive sentence *"Wishing you a very happy holiday from all of your friends at Ameristar!"* was classified as negative.

It was also suggested that the online demo performed better than the downloadable model. This could not be verified, as the online demo has likely not been operational for the last two years or more. Nevertheless, a few comments suggested that the model was undertrained and when retrained for 24 hours using the supplied dataset, the quality of classification improved, correctly classifying the abovementioned sentence as positive. We attempted to follow this advice. When training a new model, one can choose the number of tensors to be used, implying larger or smaller models. However, we were unable to find a way to increase the number of training iterations that is limited to 1000, nor were we able to extend the 24 hour training limit, which is said to be hardcoded in the application. Also, there likely is a stopping criterion applied that cannot be altered. This criterion stopped the training of smaller models we attempted to create after $300 - 400$ iterations when the model reached around $78\,\%$ classification precision on the Stanford Sentiment treebank data. Consequently, this training did not have major consequences on the precision when tested against the twitter dataset, which was as low as 17% in the initial experiments.

Multiple improvements were made to the data before the use of TNNs to improve the performance. The TNN classificator provides sentence-level classifications, but some tweets consist of multiple sentences. Initially, the sentiment of multiple sentences was aggregated by average – if the majority of the sentences was positive, the whole tweet was considered positive, if the number of positive and negative sentences was equal, the tweet was considered neutral and negative otherwise. However, it was later observed that the longest sentence usually carried the main message (since a tweet is limited to 140 characters, there is not much space for multiple elaborate sentences) and the shorter sentences were usually only short exclamations to underline the main sentence. Consider, for example, this sentence.

Due to all the rain there may be no Red Sox – Yankees gam & that sucks. Excuse my language kind sir.

The overall sentiment is clearly negative as the main concern is the game. This is expressed in the first sentence. But the second one is positive, in spite of it being only an apology for the harshness of the first one.

Next, twitter-specific constructs and hypertext links were modified, as these are common on twitter and very rare in the Stanford Sentiment Treebank the classifier was trained on. Hyperlinks and @-references were removed, since both groups are unlikely to contain a sentiment expression relevant to the tweet. #-prefixed words, on the other hand, can express keywords as mentioned and so only the # sign was stripped in order for the classifier to treat them as normal words if possible.

Using these adjustments, 41% precision was achieved. The next improvement was the consideration of emoticons. These tend to give a very good clue about the sentiment and can, in fact, be used for sentiment classification independently of the sentence. When present, the only case when these signs point the classifier in the wrong direction is that of sarcasm, which is a concept difficult to capture by any approach. Simply based on observation, a list of 17 positive and negative emoticons was composed. If an emoticon was present in the tweet, the sentiment would be based on the emoticon alone, overriding the previously computed sentiment score of the pivotal sentence. This led to classification precision of 58%, which is the best precision we were able to achieve without adding a positive bias to the resulting classification.

While analysing the errors the classificator made, apart from the negative bias discussed earlier, we found some truly puzzling instances. Consider this sentence:

"Awesome wedding shoot in Amsterdam today, excited fot the next one on Saturday in a castle in Utrecht... previews coming soon...."

The sentiment is clearly positive, but the TNN classified it as being negative. After numerous experimental modifications, we found the cause – the spelling error in "excited fot". We corrected it to "excited for" and the sentence, after which the sentence was correctly classified as positive. Another, perhaps more easily understandable, example could be this:

"Im bringing the monster load of candy tomorrow, I just hope it doesn't get all squiched"

There were other cases of false negative sentiment even after we accounted for slang and misspellings. For example, the word "just" was associated with negative sentiment. This is true for some uses of the word, but not all. In some cases, *just* can be an amplifier of positive sentiment ("just can't wait", "just perfect"). *Just* can also express the recency of an event, in which case it bears no sentiment at all. One possible explanation could stem from the fact that the tensor network was trained on a different type of data. Movie reviews, even when written by a non-professional, tend to be more formal and use more standard vocabulary than tweets. So, although the network was trained on a large variety of sentences, learned to recognize a vast vocabulary and could cope with complex structures, it was trained for a different area, or rather for a different degree of formalism and correctness in terms of the use of syntax, although it was still using data from internet discussions, which one could expect to demonstrate a similar set of features to that of social networks. The obvious idea is to retrain the network using twitter data. However, this would be quite a laborious task considering the network requires the PTB format for training [16]. This format essentially captures sentiment trees as shown in Figure 1 and Figure 2. The parse trees can be generated by the Stanford parser, but manual annotation is needed for each of the nodes of the tree. Apart from the time consumption, a native speaker would be needed to provide reliable annotation.

Some misclassified tweets would require a cross-sentence analysis in order to evaluate the sentiment correctly. Here is an example of a tweet classified as positive by the human judges

Reading my 35 pages of my Ireland scribes[2] *tonight. Wow, I miss it. #irelandforever Now to start scheming how it will happen again...*

It is positive, because the author is brought back to the joyful memories of Ireland when reading their scribes. Also they are considering going there again. However, from a more mechanical point of view, the first sentence states a neutral fact, followed by an expression of negative emotion related to something mentioned in the first sentence and at the end there is another neutral statement. And so the classifier estimated this tweet to be negative.

On the other hand, simpler improvements could be made to cover tweets such as the following.

Dallas!!! Are you ready to come have a great night with us?! We can't wait to blow the roof off with you!

Again, a false negative classification, in this case caused by the rather violent last sentence, which becomes positive only when properly combined with the previous one, making it a vivid expression of joy. Generally, the expression „can't wait" is often problematic, being usually related to a positive anticipation considered negative by many machine learning-based classifiers. Such phrases could be the target of a pre-processing phase that would exclude them or convert them to a clearer alternative. However, this would be a shift from machine learning to lexicon-based methods, as mentioned at the beginning of this chapter. In fact, we

[2] manuscripts

already made a step in the direction of a lexicon-based method by handling the emoticons separately and calculating a sentiment score.

3.4.Summary

In the context of social networks, tensor networks did not perform as well as anticipated. It may have been caused by a number of factors. Besides the potential undertraining of the model, the model showed to be easily confused by misspellings and the use of negative words in a positive way (“monster load of candy”), where the sentiment of the components of the sentence was not composed correctly. Emoticons and #-notations that often occur in tweets, can also be considered a form of misspellings the model was not prepared for. Generally, the model was to an extent dependent on the sentences being well-structured and syntactically correct, since the tensors navigated along a syntax-based parse tree. This tree often cannot be formed correctly, as the content of social networks and other internet sites often lacks standard literary structure. It is therefore questionable whether language-based methods are appropriate for the data of social networks, even when created for the same language. If these methods are to be used, their robustness to various types of mistakes the users make and unconventional expression they use should be considered when designing such a method.

In the next chapter, we discuss the simpler, language-independent model of support vector machines.

4. Support vector machines

Support vector machines are a supervised learning model. The basic idea of the model is as follows. Given a set of instances $(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \ldots, (\boldsymbol{x}_n, y_n)$, $\forall i: \boldsymbol{x}_i \in \mathbb{R}^d, y_i \in \{-1,1\}$, where the vectors $\boldsymbol{x}_i$ represent the attributes of each instance and the indicator variables y_i divide the instances into two classes, labelled -1 and 1, we search for the ideal boundary between the classes. The boundary is a hyperplane in the attribute space $\mathbb{R}^d$ that separates the two classes of instances. Ideal here means having the largest separation margin – the distance of the ideal separating hyperplane from the closest instance $\boldsymbol{s}$ is the largest possible.

Support vector machines are similar to the logistic regression model discussed in Chapter 2 – both models separate the instances belonging to different classes by some boundary. However, SVMs are more focused on the boundary and its immediate surroundings than logistic regression. While logistic regression takes all the instances into account with equal importance when computing the log-likelihood, the boundary of a SVM is defined by the closest, marginal instances from each class. The normals to the boundary hyperplane on each side "support" the boundary, hence support vector machines.

SVMs are a rather geometric model, whereas the logistic regression is a probabilistic classifier, calculating the probability of an instance belonging to a certain class. SVMs, in their basic form, search for a linear hyperplane providing a clear separation and can be less efficient when applied to problems where the classes are not linearly separable. Therefore, SVMs are more likely to succeed in high-dimensional spaces with less noise.

However, SVMs can still handle non-linearly separable problems when the input instance space is transformed and expanded appropriately, while maintaining the properties of the original space that are needed for SVM classification. These techniques are known as kernel methods and are further discussed in Section 4.2.

4.1. Basic classifier

We will now focus on binary classification problems, which are the basic task for SVMs. A multinomial classifier can be composed by training a binary classifier for each pair of classes and assigning the class that wins the most comparisons.[3]

A hyperplane in $\mathbb{R}^d$ can be defined as the set of $\boldsymbol{x} \in \mathbb{R}^d$ satisfying $\boldsymbol{x}\boldsymbol{w} + b = 0$, where $\boldsymbol{w}$ is the normal vector of the hyperplane and $\frac{|b|}{\|\boldsymbol{w}\|}$ is the distance of the hyperplane from the origin. By $\boldsymbol{w}\boldsymbol{x}$ we denote the dot product of the two vectors, or, in matrix notation, $\boldsymbol{w}^T\boldsymbol{x}$. This will be the case through this chapter. The marginal instances lie on hyperplanes parallel to the ideal separating hyperplane.

[3] Other, more efficient, multinomial classification methods exist, though [54]. These were used for multinomial classification in our experiments, too. We do not explain this method in detail here, as it is not particularly associated with SVMs.

We can express the marginal hyperplane on the border of the $+1$ class as $\boldsymbol{wx} + b = 1$ and the hyperplane on the border of the -1 class as $\boldsymbol{wx} + b = -1$. If we want to prevent any instances from occurring closer to the boundary than the margin planes, we can establish the following constraints, applied to each instance and try to find the boundary hyperplane $\boldsymbol{wx} + b = 0$ for which these constraints hold:

$$y_i = 1 \Leftrightarrow \boldsymbol{wx}_i + b \geq 1, \tag{17}$$

$$y_i = -1 \Leftrightarrow \boldsymbol{wx}_i + b \leq -1 \tag{18}$$

After we have found the ideal separating hyperplane characterized by its normal vector $\boldsymbol{w}$, we can classify a new instance $\boldsymbol{x}'$ by calculating

$$q = \boldsymbol{wx}' + b \tag{19}$$

If $q > 0$, the estimated class of $\boldsymbol{x}'$ is $+1$, if $q < 0$, it is -1. The application of the conditions (17) and (18) is show in Figure 3.

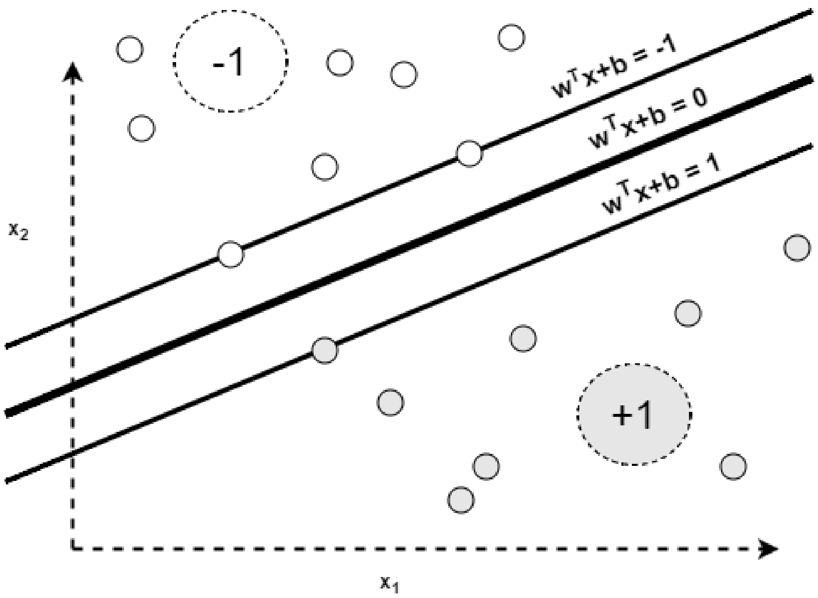

FIGURE 3: A TWO-DIMENSIONAL SVM EXAMPLE WITH LINEARLY SEPARABLE DATA. THE BOLD LINE IS THE BOUNDARY SEPARATING THE TWO CLASSES.

These conditions can be satisfied for all instances only if the data is linearly separable. To account for datasets where this is not the case, we introduce an error function, also known as the hinge loss function:

$$h(\boldsymbol{x}) = \max\big(0, 1 - y_i(\boldsymbol{wx} + b)\big) \tag{20}$$

The function equals 0 when the instance is classified correctly. When the instance is misclassified, the value of the function is proportional to its distance from the margin. This approach (called the soft-margin, since it allows errors) gives us a good basis for the error we want to minimize. An overall error function should, however, also force the classifier to find the best separating hyperplane as defined earlier, by maximizing the margin, the size of which is $\frac{2}{\|\boldsymbol{w}\|}$, as explained in Figure 3. When the data is not linearly separable or the margin is small due to outliers, we need to achieve a reasonable trade-off between the margin and the classification error. The overall error function we want to minimize will be

$$\begin{aligned} E(\boldsymbol{w},\lambda) &= \frac{1}{n}\sum_{i=1}^{n} h(\boldsymbol{x}_i) \ + \ \lambda\|\boldsymbol{w}\|^2 \\ &= \frac{1}{n}\sum_{i=1}^{n} \max(0, 1 \ - \ y_i(\boldsymbol{w}\boldsymbol{x}_i + b)) \ + \ \lambda\|\boldsymbol{w}\|^2 \end{aligned} \tag{21}$$

Here, $\boldsymbol{w}$ is the vector defining the hyperplane to be optimized and λ is the parameter regulating the significance of the size of the margin and the classification error on training data. A large value of λ implies a larger margin, whereas setting $\lambda = 0$ allows the classifier to shrink the margin as needed if it separates the data. By allowing for classification errors in the soft-margin problem and using the hinge loss function, we can formulate a variation of the conditions (17) and (18) that the optimal solution of (20) must satisfy.

$$\forall i: y_i(\boldsymbol{w}\boldsymbol{x}_i + \ b) \geq 1 - h(\boldsymbol{x}_i) \tag{22}$$

By using Lagrange multipliers $\boldsymbol{c} = (c_1, \dots c_n), \forall i : c_i \in \mathbb{R}$ as explained in [17], we can derive the dual problem

$$\max_{\boldsymbol{c}} g(\boldsymbol{c}) = \sum_{i=1}^{n} c_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} y_i c_i \langle \boldsymbol{x}_i, \boldsymbol{x}_j \rangle y_j c_j \tag{23}$$

where $\langle \_ , \_ \rangle$ denotes the inner product. The corresponding Karush–Kuhn–Tucker conditions, that need to be fulfilled in order for the solution to be optimal, follow.

$$\sum_{i=1}^{n} c_i y_i = 0, \forall i: 0 \leq c_i \leq \frac{1}{2n\lambda} \tag{24}$$

Our new goal is the maximization of $g(\boldsymbol{c})$ while the conditions in (24) must be fulfilled. This problem is more feasible, especially when kernel functions are applied, as we will see in the next section.

4.2. Kernel functions

As mentioned earlier in this chapter, SVMs work best when the data is linearly separable and they can ensure a large margin between the two classes. The problem is, how to cope when the data is not linearly separable. One simple solution is adding more features so that the classes become separable in a higher-dimensional space. For example, we could add polynomial combinations of the attributes and transform the set of features

$$a, b \tag{25}$$

into

$$a, b, a^2, ab, b^2 \tag{26}$$

Generally, we use a feature-mapping function $\Phi\colon \mathbb{R}^d \to \mathbb{Q}$, where $\mathbb{Q}$ is a vector space, usually of a higher dimension than d. By using radial basis functions, we can even map the original attribute vectors into an infinite-dimensional space. Dramatically increasing the dimension of the vector space, however, leads to obvious computational problems. The transformation into the high-dimensional space is costly in itself. The same is also true for the subsequent operations with the expanded vectors.

However, when solving the dual problem, we do not need to know the values of the components in the expanded vectors – we only need to work with their pairwise inner products. If we were able to compute the inner products of the transformed vectors in a fast manner, we could increase the performance significantly by avoiding the necessity to perform the actual data transformations as defined by the kernel function. Such an option is provided to us by the so-called kernel functions. A kernel function for the feature-mapping function $\Phi : \mathbb{R}^d \to \mathbb{Q}$ is defined as

$$k(\boldsymbol{x}_1, \boldsymbol{x}_2) = \langle \Phi(\boldsymbol{x}_1), \Phi(\boldsymbol{x}_2) \rangle_{\mathbb{Q}} \tag{27}$$

The most commonly used feature transformations have kernel functions that are easy to compute. For example, for

$$\Phi(\boldsymbol{x}) = \Phi((a,b)) = \left(a^2, b^2, \sqrt{2}a, \sqrt{2}b, \sqrt{2}ab, 1\right) \tag{28}$$

we have

$$\begin{aligned} k(\boldsymbol{x}_1, \boldsymbol{x}_2) &= \langle \Phi(\boldsymbol{x}_1), \Phi(\boldsymbol{x}_2) \rangle = (\boldsymbol{x}_1 \boldsymbol{x}_2 + 1)^2 \\ &= (\langle (a_1, b_1), (a_2, b_2) \rangle + 1)^2 \\ &= (a_1 a_2 + b_1 b_2 + 1)^2 \\ &= a_1^2 a_2^2 + b_1^2 b_2^2 + 2a_1 a_2 + 2b_1 b_2 + 2a_1 a_2 b_1 b_2 + 1 \\ &= \langle \left(a_1^2, b_1^2, \sqrt{2}a_1, \sqrt{2}b_1, \sqrt{2}a_1 b_1, 1\right), \left(a_2^2, b_2^2, \sqrt{2}a_2, \sqrt{2}b_2, \sqrt{2}a_2 b_2, 1\right) \rangle \end{aligned} \tag{29}$$

This can be generalized to higher-degree polynomials where the kernel function takes the following form for a polynomial of degree p:

$$k(\boldsymbol{x}_1, \boldsymbol{x}_2) = (\boldsymbol{x}_1 \boldsymbol{x}_2 + 1)^p \tag{30}$$

When we apply a kernel function, the optimization problem of (23) can be rewritten as follows, while the KKT conditions of (24) hold:

$$\max_{\boldsymbol{c}} g(\boldsymbol{c}) = \sum_{i=1}^{n} c_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} y_i y_j k\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right) c_i c_j \tag{31}$$

where the Lagrange multipliers $\boldsymbol{c} = (c_1, \ldots, c_n)$ are the parameters to be optimized.

4.3. Sequential minimal optimization of SVMs

The optimization problem stated in (31) is a quadratic programming problem involving an $n \times n$-dimensional matrix where n is the number of training instances. Solving this problem requires large amounts of memory and processor time already for thousands of training instances, making training SVMs very difficult.

Multiple methods have been developed in order to reduce the size of the QP problem. One of them that was popular prior to Sequential minimal optimization is known as "chunking" [18]. This method is based on the fact that the value of the quadratic form in (31) does not change if Lagrange multipliers that equal 0 are excluded. This can reduce the size of the problem significantly, but the non-zero multipliers need to be identified first. This is done by iteratively adding multipliers to a growing set until there are no examples in the dataset that violate the KKT conditions. While this reduces the amount of memory needed to store the matrix describing the QP problem from n^2 to the square of the number of non-zero Lagrange multipliers, this can still be a problematic volume for large datasets.

Platt used the theorem of [19] which states that the SVM QP problem can be solved via a series of smaller subproblems as long as at least one training instance $c_i, i \in \{1, \dots, n\}$ that violates the KKT conditions is added into consideration at each step. Under these conditions, each subsequent step will reduce the value of the objective function and maintain a feasible solution.

Platt reduced the number of Lagrange multipliers optimized in each step to the minimum, 2. This means that the partial problems can be solved analytically and the storage of the QP matrix can be avoided entirely.

When only two elements of the $\boldsymbol{c}$ vector of Lagrange multipliers change and the conditions of (24) still apply, one of the multipliers must increase proportionally to the decrease of the second one and the shift is limited by the fact that the resulting values must fall into the interval $\left[0, \frac{1}{2n\lambda}\right]$.

SMO also uses heuristics to choose the Lagrange multipliers which to optimize first in order to reduce the number of times the KKT conditions have to be checked for each multiplier. The algorithm searches for Lagrange multipliers that violate the KKT conditions. The heuristic used preferably checks the KKT conditions for instances that are "non-bound" – their value is neither 0 nor $\frac{1}{2n\lambda}$, as they are more likely to violate the conditions. The set of non-bound multipliers is stored in the memory and optimized repeatedly until all its members obey the KKT conditions. Then the entire vector $\boldsymbol{c}$ is checked again.

Benchmarks performed by Platt showed that the SMO algorithm was approximately $100 \times$ faster than the chunking method, in some cases reaching even $1000 \times$ the speed of chunking. Therefore, this algorithm will be used later in this and the following chapters.

4.4. Experiments

To test the performance of support vector machines, we reused parts of the text preprocessing developed in previous chapters, mainly in Chapter 3. This preprocessing included removing user names (which are prefixed with "@" in Twitter posts) and topic references (prefixed with "#"). URL addresses and some punctuation of no added value was also removed. Emoticons, on the other hand, were kept in the text. We used $1-3$-grams by default and from the ngrams occurring in the collection, the top 1000 were chosen based on their correlation with the target class (sentiment) as the input attributes for the SVM algorithm. The model was trained using the Platt's method described in Section 4.3.

The data originated from the SemEval 2013, task 2 and was the training set of 4671 instances. 3088 of them were used for training and the remainder served as the test set. Using the same dataset with the same preprocessing, other models were trained to be compared with the SVM. The comparison can be seen in Table 5.

Method	**Precision**
Naïve Bayes	84.8866 %
Logistic regression	90.6171 %
SVM	91.6877 %
Multilayer perceptron[4]	84.3829 %

TABLE 5: *COMPARISON OF COMMON LEARNING ALGORITHMS ON THE SEMEVAL 2013 TRAINING DATASET*

These results were rather optimistic, which can be caused by the fact that the training set had been cleaned by the organizers of SemEval and some instances could have been excluded to ease training of models.

Nevertheless, this test is sufficient for the purpose of comparing the accuracy of different models. Based on the results, SVMs are definitely among the perspective models.

4.5. Summary

SVMs are one of the most common techniques used for supervised learning in the field of text classification. They seem to be the slightly better option when compared to logistic regression discussed in Chapter 2.

In Chapter 5, we will discuss and test options for analysing emoticons and slang. SVM and logistic regression will be among the referential models used.

[4] A standard backpropagation neural network with two hidden layers, each containing 100 neurons, and 100 input neurons corresponding to the 100 attributes with the highest correlation with target class. Using this configuration, the time needed to train the model was comparable to the other models. Higher precision was achieved when more neurons were used in the input and hidden layers, but with increasing networks size the model quickly becomes computationally unfeasible. This is further discussed in Chapter 7, which proposes a more efficient high-precision solution using convolutional neural networks.

5. Analysing slang and phraseology

In this chapter, we develop further on the observations made in Section 3.3. In that section, the positive phrase "*blow the roof off with you*" is used to demonstrate the kind of phrase that poses a great difficulty the tensor-based neural network model. In fact, this kind of phrase would be problematic in most systems that perform syntactic and semantic analysis, even of superb quality, as long as the systems would only consider the literal meaning of the words and the structure of the sentence.

Generally, phrasal expressions can often be puzzling even to a human being, unless they have heard the expression before or it is derived from a similar one that they are familiar with. Expressions such as "*cost an arm and a leg*", "*break a leg*" or "*to feel blue*" do not indicate the actual meaning even remotely.

While these expressions are difficult to interpret, we cannot afford to disregard them, especially when analysing sentiment in the context of social networks or internet forums, since these consist mostly of very short texts in which phrasal expressions can often be the only sentiment indicator available to us.

Furthermore, we need to account for slang terms and misspellings. To simplify, we could consider misspellings as a sort of slang, as these are often used intentionally to make the expression more interesting or to emphasize it ("*greaaaat*"). Slang words are closely related to phrasal expressions in terms of the means we can apply to understand them. As is the case with multi-word phrasal expressions, slang words will often be difficult to understand, even to a human, unless they already know their meaning (such is the case with standard words as well). Also, slang words are atomic – the meaning cannot be derived from analysing their syllable or letter components. And so are also idiomatic expressions – understanding the words they consist of does not help in understanding the whole, as demonstrated in previous expressions.

When compared to slang words, multi-word phrasal expressions face us with additional difficulty stemming from the fact that each of the words these phrases are formed by can have a meaning of their own and so an incomplete idiom cannot be considered a misspelling, on the contrary, it is likely to be a part of normal speech that should be evaluated using standard measures. Also, in some sentences a sequence that corresponds to a phrasal expression occurs, but the phrasal expression is in fact not used. For example:

She <u>put</u> the coat <u>on</u> the table.

In this sentence, the post-particle is vital in order to identify that the phrasal meaning of "*put on*" was indeed not used.

Furthermore, phrasal verbs are usually separable in the syntax – noun phrases can be, but are not necessarily, inserted in the expression. For example:

He turned off the radio.

He turned the radio off.

So far, this points towards the necessity for an extensive sentiment lexicon of phrasal and slang terms. However, yet another complication can be identified when considering how much the vocabulary and phrases have changed in the past few years. In the past twenty years, the tremendous technological advances have completely transformed the way of communication, especially for the young generation. Terms related to technology, such as "*to google something*", that were virtually unheard of until the 21st century, are now globally understood. This trend is likely to continue and therefore, the composition of a sentiment lexicon covering these terms will likely require the ability to adopt new terms dynamically and, in the ideal case, automatically.

It is also important to consider that phrasal expressions represent a large part of speech, while we are interested only in those carrying sentiment. In the English language, phrasal verbs contribute as much as one third to the total verb vocabulary [20], while only a fraction is relevant to sentiment analysis. Therefore, an all-covering lexicon is not necessary, since the lexicon is not the aim, but the means of our task.

To recapitulate, we need to solve certain issues related to slang and phrasal terms, probably in the following sequence:

1. Identifying a dynamic source of phrasal expressions bearing sentiment.
2. Annotating the expressions in terms of sentiment.
3. Comprising a phrasal sentiment lexicon.
4. Applying the lexicon to data from the social media, while accounting for deviations from the definitions in the lexicon and still correctly using other techniques for the most of the content, that is not phrasal.

In the following sections, we will look into possible approaches to these tasks.

5.1. Related work

In [20], an expert lexicon system was presented with the aim of identifying and parsing phrasal verbs. The system achieved a remarkable F-score[5] of over 96%. The system uses a set of rules (macros) that identify three basic types of phrasal verbs. While this does not generalise well to the overall task of phrasal and slang analysis, the method proposed, besides performing well in the restricted domain, divides the tasks of identification and parsing of phrasal verbs, the reason being that the parsing can be performed by a general native language parser after the phrasal verb has been identified. This is the case of most compound phrasal expressions.

[5] F-score is defined as the harmonic mean of the precision and recall measures, typically computed from the confusion matrix of a binary classifier.

[21] took a similar approach with a local grammatical rules identifying phrasal verbs with the exception that their grammar was much simpler and was based on a freeware general-purpose linguistic development environment, NooJ (http://nooj4nlp.net), that has been developed by Max Silberztein since 2003. The accuracy is slightly lower, achieving only 84% on average, but the method is more suitable for extensions that can cover other aspects of phrasal analysis.

While these works have only focused on a part of the problem and did not specifically deal with sentiment analysis, a recent study by [22] aimed directly at sentiment classification of tweets, using data from SemEval 2013. As well as the data, some of the techniques employed were similar to those conducted earlier in section 3.3. A special dictionary was used to extract the sentiment from emoticons (the dictionary was significantly larger than that used in 3.3). Twitter-specific marks such as “@” and “#” were labelled during pre-processing. Apart from that, a slang dictionary was used to label general and Twitter-specific slang properly. The model was created using machine learning with support vector machines with n-grams (mono- and bigrams) as attributes, similarly to 2.3. The resulting precision achieved the remarkable figure of 85%. The model did intentionally not use a deeper syntactic analysis in order to maintain language independence. In the following sections, we combined some features of this approach with the experience from previous chapters and observations about the data at hand. A few models were evaluated to test the effectiveness of our approach.

5.2. Emoticon and slang dictionaries

We used the [23] emoticon list to form a dictionary of 232 emoticon expressions. We also formed a dictionary of commonly used slang with 300 entries from [24]. The slang dictionary, previously known as chatslang.com, now offers thousands of slang terms divided into multiple categories based on the areas they usually occur in. Since we intended to use this dictionary for analysing Twitter data, we used only the collections of general and Twitter-related slang. The slang dictionary showed to be more difficult to process, as most of the expressions were short lowercase abbreviations that could easily be confused with proper words or their misspellings. The emoticons, on the other hand, were clearly distinguishable and could be identified in the tweets with high precision, since most of these use characters that do not occur in normal speech.

Moreover, the entries in the considerably smaller emoticon dictionary were identified in over 40% of the tweets used in experiments, while the slang dictionary found use in only over 21% of the entries.

The emoticons also have clear interpretations and are rarely misleading. The tweets in which emoticons were present, could be classified as positive or negative based purely on the emoticons with a reasonable 65% precision. The use of slang translations is more difficult as the expressions are not necessarily emotional and have to be evaluated as a part of the sentence or tweet. Simply replacing the expression with its definition can also disrupt the structure and meaning of the sentence.

Due to these difficulties associated with slang translation and the limited potential performance boost they offered, we excluded the translation of slang (not emoticons) from further experiments. Still, slang is often part of posts on social networks. Many of these expressions are in fact abbreviations and their purpose is simply to shorten the proper version of a word or group of words. This means that slang may not actually be the correct term here and these abbreviations could be understood as a compressed vocabulary by which the language was adapted to the constrains of social networks. Learning this new language could then be more suitable than translating it. These considerations influenced the choice of tested machine learning methods.

5.3. Experiments and results

In chapter 3 we saw that recursive tensor networks can find it difficult to identify and compose the sentiments of words in sentences that do not follow standard structures. In section 5.2 we further discussed how the shortening of words and expressions disrupts these structures and how the translation of slang can in fact worsen this, although it may replace the slang word (or abbreviation) with a more standard sequence of words. We also observed in chapter 2 that logistic regression achieves reasonable results. Logistic regression and other "bag-of-words" methods have one advantage over the methods that depend on POS tagging and sentence structure parsing in general – they can adapt to any vocabulary and any conventions naturally. This advantage grows larger as the texts become more informal and less structured, as more misspellings occur and special sequences of characters are used that would not be covered by any "serious" dictionary.

Based on this, we chose methods that did not depend on the syntactic structure, both simple and more complex ones, that could show whether a non-syntactic approach would perform better than a more complex tagger-dependent method.

We also applied part of the observations made in chapter 3 concerning the preprocessing that improved the performance of tensor neural networks, such as removing "#" and "@"-prefixed words and URL addresses. The logistic regression model, although discussed and tested in chapter 2, was thus used again, mainly for comparison to the other models.

In all, these 4 models were tested:

- Multinomial Naïve Bayes classifier
- Support vector machines
- Hidden Markov models
- Logistic regression

The dataset used once again originated from SemEval, in this instance the data from SemEval 2013, task 2, subtask B were used. This dataset contains 4671 tweets labelled as positive, neutral and negative.

The set of attributes for each of the models, except for hidden Markov models, consisted of $1\,000$ $1-3$ –grams (this is a compromise between the $1-5$ –grams used in chapter 2 and $1-2$ –grams used by [22]) ranked based on the expression's correlation with the output (the correct sentiment class). The ngrams were also stemmed using the iterative Lovins algorithm and stopwords were filtered. Some of the stemmed attributes with the highest correlation are shown in Table 6 below.

Word	**Correlation**
good	0.1533
hap	0.1359
excit	0.1234
great	0.1208
lov	0.1172
t	0.1128
it	0.10287
best	0.107
the best	0.1028
not	0.1017
can t wait	0.0988
you	0.097
wait	0.095
awesom	0.093
so	0.0919
a great	0.0893
i	0.0874
is	0.0826
fun	0.0809
happy v	0.0804
can t	0.0804
thank	0.0801

Table 6: *The stems of 1 - 3-grams from the SemEval 2013, task 2B train dataset with the highest correlation with the output class.*

As mentioned before, the scores from emoticons are a valuable indicator of sentiment in tweets and so it was included as an additional attribute in the input with the values *emotes_0*, *emotes_positive* and *emotes_negative*. The emoticons were also kept present in the tweets. The reasoning behind this was that since the methods did not depend on any particular syntax, they could pick up the emoticons (as well as slang) as a part of the vocabulary and naturally consider the most frequent ones in their estimations.

As for the hidden Markov models, the format of the input was necessarily different because the model analyses sequences rather than sets of attributes. It was discovered that 95% of the tweets had 28 words or less and so the length was normalized to this, trimming the longer tweets and padding the shorter with "__EMPTY__" words. The vocabulary consisted of 1000 words (1-grams) and these were chosen based on their overall frequency in the dataset rather than the correlation. When a word occurred in a tweet that did not qualify into the vocabulary, it was replaced by a special "__UNKNOWN__" word. The main reason for this was the attempt to make minimal changes to the sequence of words in the tweet. Each word that was included in a model's training instance remained at its correct ordinal position and the frequency criteria for choosing the vocabulary was used so that as few words as possible would be replaced with "__UNKNOWN__".

With all the tested models, we observed a slight improvement in the classification accuracy after the emoticon scores were added to the attributes. However, this improvement was statistically insignificant. For example, the accuracy of the SVM classificatory improved from 89.4% to 89.9%, which clearly does not prove the benefits of adding the sentiment score to the attributes.

On the other hand, since emoticons are relatively frequent in social network posts and are quite clear indicators of sentiment when present, a classificator achieving up to 60% accuracy could be constructed without using any machine learning method. In this case, the tweets that were tagged by *emotes_positive*, had their class estimated as positive and the ones tagged by *emotes_negative* were classified as negative.

As for the translation of slang phrases, as noted in 5.2 a dictionary of 300 common slang expressions was used, however, only in 21% of the twitter dataset a match could be found and the dictionary applied. This implies that the influence of slang translation was rather insignificant and in some cases even led to slight performance deterioration.

5.4. Summary

Although slang words and especially emoticons are a common new phenomenon in the content of social networks and internet discussions in general, the results of our experiments did not prove that treating these expressions specifically brings a significant improvement. This can be explained by the nature of the training models we used. This choice was not arbitrary, and was mainly based on the experiments performed in Chapters 2 and 4, which showed that relatively simple bag-of-words models that do not depend on syntactic structures can perform well.

The bag-of-words models do not analyse the content from a linguistic standpoint. This also means that the models can "approach" any language or pseudolanguage, if we may refer to the content of social networks in this manner, equally. Emoticons and phrasal expressions can be learned similarly to standard words, if they appear frequently enough and are a statistically significant indicator of the target class the classificator model is trained for.

6. Clustering

The basic task of cluster analysis is the discovery of groups among instances of a given dataset that share common properties and the description of and distinction between these groups. In this chapter, we discuss some of the techniques used for clustering and how these methods can be used in our main task of sentiment analysis.

Clustering, unlike the methods that have been described in the previous chapters, uses unsupervised learning. Unsupervised learning creates a model of the data based on a metric of its quality. This quality, however, is not defined by the user, but is rather based on some general metric of "closeness" of the model to the input data. This means that the metric the algorithm used may not be directly related to the task we are working on. The hypotheses formed by unsupervised learning are therefore relatively open to interpretation. The algorithms we will use form clusters based on the attributes of the instances provided, but it is our task to describe the resulting clusters and their meaning with regard to sentiment. On the other hand, even if the clustering method created a result that is not related to sentiment, it can provide us with some insight into the data that can help in further improvements to other algorithms we will be using.

6.1. K-means clustering

As an introduction to clustering, we briefly describe one of the most popular clustering algorithms, known as k-means. This algorithm arranges k centroids, or vectors in the attribute space, in a way that minimizes the distance of each centroid to the training instances to which this centroid is the closest one, usually in terms of the Euclidian distance.

More precisely, given a training dataset D consisting of n vectors in the d-dimensional feature space V and the set of centroids $C = \{\boldsymbol{c}_i, \boldsymbol{c}_2, \dots, \boldsymbol{c}_k\}$, the closest centroid is determined for each training instance $\boldsymbol{x}$. This can be expressed using a clustering function that finds the index of the closest centroid to a given input vector $\boldsymbol{x}$.

$$c(\boldsymbol{x}) = i : \forall j \in \{1, \dots, k\} : S(\boldsymbol{x}, \boldsymbol{c}_i) \leq S(\boldsymbol{x}, \boldsymbol{c}_j) \quad (32)$$

Here, S stands for the distance metric.

We will describe the Lloyd's algorithm [25], which is most often used for k-means. The training of the model is performed iteratively. In each iteration, the training instances are first reassigned to the nearest centroids and then the positions of the centroids are adjusted to lie in the centre between the instances assigned to them. The process is then repeated until the positions of the centroids stabilize or until a pre-set number of iterations is reached.

In the first step, we can use the clustering function $c(\boldsymbol{x})$ to divide the dataset D into k mutually disjoint subsets $D_1, D_2, \dots, D_k$ corresponding to the centroids, where $D_i = \{\boldsymbol{x} \in D : c(\boldsymbol{x}) = i\}$.

In the second step, the new position $\boldsymbol{c'}_i$ of each centroid $\boldsymbol{c}_i$ is determined as the average of the positions of the instances assigned to it:

$$c'_{i,j} = \frac{1}{|D_i|} \sum_{x \in D_i} x_j \tag{33}$$

This algorithm decreases the overall sum of distances between the instances and the corresponding centroids in each epoch and is thereby guaranteed to converge towards a stable position of the centroids. There is however, no guarantee that the clusters produced are optimal. On the other hand, finding the optimal clusters has a complexity of $O(n^{dk+1} \log n)$ for fixed n and k, which is computationally much costlier than the Lloyd's algorithm with the complexity of $O(nkdt)$ where t is the number of iterations performed [26].

6.2. Kohonen maps

Kohonen maps (also known as self-organizing maps or SOMs) are a special kind of neural networks where the neurons are organised in a topological way, usually a 2-dimensional grid or a single sequence of mutually interconnected neurons. Neighbouring neurons are interconnected and influence each other during the learning process. The reason for this is that we want the neurons that are topologically close to each other to have similar properties, to learn to recognize similar inputs and thereby represent a certain cluster in the feature space.

6.2.1. Computational model

In addition to the connection between neighbours on the topological grid, each neuron is connected to each input of the neural network, corresponding to an attribute of the training data. Each of these connections to the input has a weight $\boldsymbol{w}_{i,j} \in \mathbb{R}$ where $i \in \{1, \dots, m\}$ denotes the neuron in the grid and $j \in \{1, \dots, n\}$ the input attribute. The $m \times n$ matrix $\boldsymbol{W}$ is to be learned through iterative adjustments aimed at representing the input vectors by the weight vectors of the neurons.

The weights are initialized randomly. Then, input vectors are presented to the network one by one. When processing a certain input vector $\boldsymbol{x}$, the Euclidian distance of this vector to the weight vector of each of the neurons is computed first. The "closest" neuron in terms of this distance is chosen as the representative of the input vector. This neuron is often called the winning neuron. We will suppose here that the index of the winning neuron is e and its weigh vector is denoted by $\boldsymbol{w}_e$. To represent the input more closely, the weights of the winning neuron e are adjusted towards the input:

$$\Delta \boldsymbol{w}_e = \sigma(\boldsymbol{x} - \boldsymbol{w}_e) \tag{34}$$

Here, $0 < \sigma < 1$ determines the degree to which the weight of the neuron is changed towards this single input.

After the first pass through the input vectors, each instance of the input dataset has been "associated" with the corresponding winning neuron and the weights of this vector have been modified to better reflect the input. Some neurons could have "caught" multiple inputs and have adjusted their weights multiple times. To stabilize the learned map, the process is repeated, usually up to tens of thousands times, and each time σ is decreased slightly so that the information learned in previous passes through the dataset is not lost too easily.

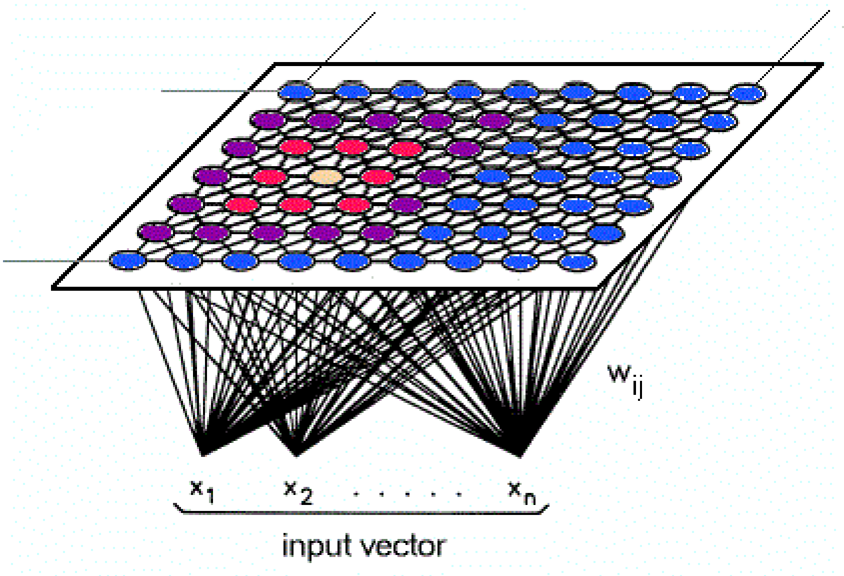

FIGURE 4: A KOHONEN MAP WITH A TWO-DIMENSIONAL TOPOLOGICAL GRID. EACH OF THE INPUT VECTORS ON THE BOTTOM SIDE IS CONNECTED TO ALL THE NEURONS (CIRCLES) IN THE UPPER LAYER, REPRESENTING THE MAP. THE BRIGHT WINNING NEURON INFLUENCES ITS NEIGHBOURS UP TO A GIVEN DISTANCE.

We will now extend our model to include also the interactions between neurons that are close to each other with regard to the topology of the map. This will force the map to reflect also the topological relationships among the input patterns. Besides adjusting the weights of the winning neuron to match the instance more closely, we will also modify the neurons in the neighbourhood of the winning neuron towards the instance. By neighbourhood we understand the neurons that are topologically close to the winning neuron on the neuron grid. This is shown in Figure 4, where the bright neuron is the winning one, being the closest match to the current input. The further a neuron is from the winning one, the less its weights are modified. In Figure 4 this means that the weights of red neurons are modified more than the weights of purple neurons.

To give these ideas a formal ground, we define the changes $\Delta \boldsymbol{w}_j$ made to the weights $\boldsymbol{w}_j$ of a neuron in response to the input vector $\boldsymbol{x}$ during the iteration t. This is a general rule that will be used for the winning neuron as well as for its neighbours. We use the function $\delta : \mathbb{N} \times \mathbb{N} \rightarrow [0,1]$ to gradually decrease the influence of the winning neuron e on neuron j with an increasing distance between e and j. Naturally, $\delta(e,e) = 1$. The σ function used previously is now also extended to lower its value slightly with each passing iteration t.

$$\Delta \boldsymbol{w}_j = \sigma(t)\delta(e,j)(\boldsymbol{x} - \boldsymbol{w}_j) \tag{35}$$

One option for the definition of δ is

$$\delta(e,j) = \exp\left(-\frac{S^2(e,j)}{\xi}\right) \tag{36}$$

where $S: \mathbb{N} \times \mathbb{N} \to \mathbb{N}_0$ is the topological distance in the network and the parameter $\xi \in \mathbb{R}$ regulates the decrease of the impact of the winning neuron with increasing distance.

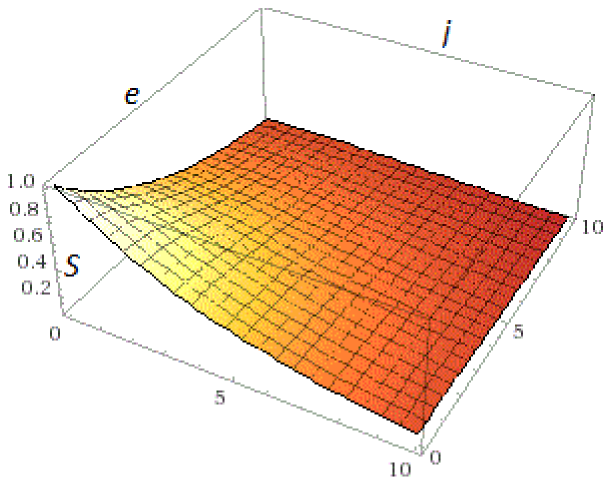

FIGURE 5: THE δ FUNCTION FOR $\xi = 5$ AND $S(e,j)$ SET TO THE EUCLIDIAN DISTANCE BETWEEN THE NEURONS ON A SOM LAYER ORGANIZED INTO A SQUARE GRID AS SHOWN IN FIGURE 4. THE FUNCTION IS PLOTTED FOR DISTANCES FROM 0 TO 10 ON EACH AXIS OF THE LAYER.

The rate of change, or learning rate, introduced in (34), now has a parameter $t \in \mathbb{N}_0$ that indicates the iteration, beginning with 0. Similarly to δ, different definitions can be used for the gradual decrease of σ. Here is one commonly used option

$$\sigma(t) = \sigma_0 \exp\left(-\frac{t}{\tau}\right) \tag{37}$$

The parameter τ regulates the speed of the decrease of the learning rate. The initial value $\sigma_0 < 1$ is also set.

6.2.2. Applications in text mining – WEBSOM

Since the introduction of neural Kohonen maps, attempts have been made to represent textual data via these maps, which promised the ability to visualize the content of multiple documents in a comprehensive and accessible way. These attempts, however, often encountered performance difficulties when the dictionary of the model became too large, resulting in a high-dimensional feature space. The training of Kohonen maps is computationally intense and the processing of a large number of documents that would provide a sufficient coverage for practical applications is often computationally too expensive.

One approach to this problem, called WEBSOM, introduced in [27], managed to cope with these difficulties. The WEBSOM system makes only minimal assumptions about the input data and, in the basic form, limits itself to a simple exploratory browsing service. The system produces an interactive map labelled by topics that occurred in documents that a SOM was trained for. The user navigates through the map, reaching more detailed clusters, labelled by common words, and eventually the document level, where they can choose a document of interest to read.

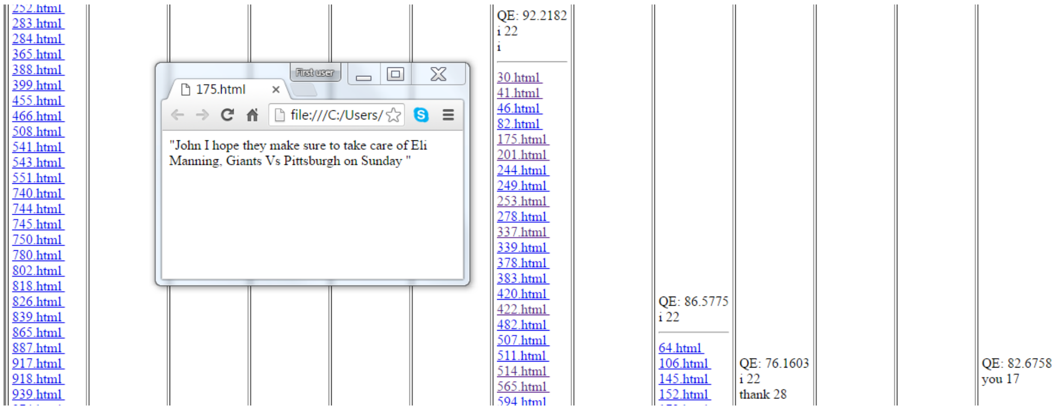

FIGURE 6: THE BOTTOM LEVEL OF AN INTERACTIVE SOM INTERFACE, SIMILAR TO THAT OF WEBSOM. THIS MAP WAS CREATED USING THE SOM IMPLEMENTATION OF THE SOMLIB DIGITAL LIBRARY PROJECT OF THE VIENNA UNIVERSITY OF TECHNOLOGY. THE GHSOM ALGORITHM (GROWING HIERARCHICAL SOM) ALGORITHM WAS USED. THIS METHOD ADDS NEW LAYERS TO THE MAP AND EXTENDS EXISTING LAYERS BY NEW NEURONS UNTIL THE MODEL FITS THE DATA CLOSELY ENOUGH. THE MAP WAS TRAINED USING THE SEMEVAL 2013 DATASET. FOR DETAILS OF THE IMPLEMENTATION, SEE SECTION 6.2.4. CLUSTERS ARE DISPLAYED AS CELLS IN A SIMPLE HTML TABLE. LINKS TO THE CLUSTERED ITEMS ARE PROVIDED UNDER THE KEYWORDS REPRESENTING EACH CLUSTER.

The authors of WEBSOM have made a significant effort to reduce the dimensionality of the instances that are used to train the Kohonen networks. A two-stage processing is applied. In the first phase, a SOM is formed on the word level in order to reduce the vocabulary to categories. The trained word map is then used to encode the contents of documents into vectors that are then used to train a second SOM on the document level, which is the final product. The architecture of the system is shown in Figure 7.

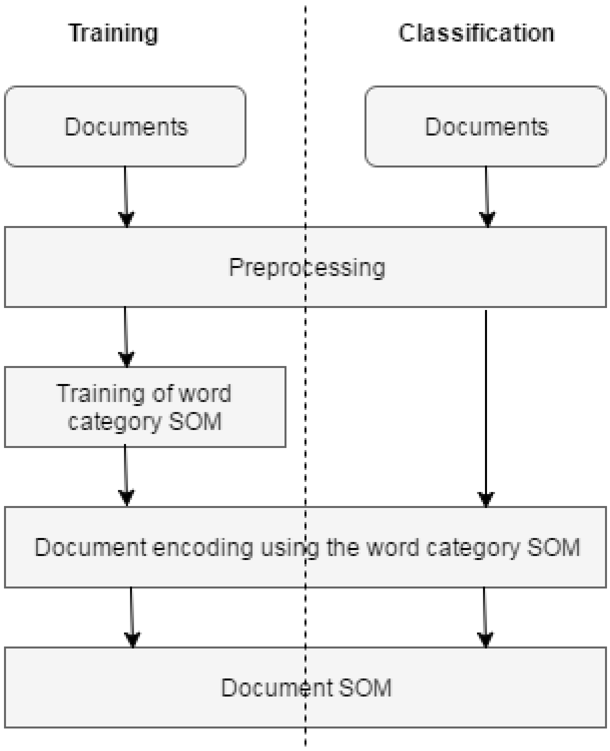

FIGURE 7: THE OVERALL ARCHITECTURE OF WEBSOM - THE SYSTEM USES A TWO LEVEL HIERARCHY OF SOMS WHERE THE SECOND LEVEL OPERATES ON THE OUTPUT OF THE FIRST ONE.

Scalability is one of the main aims of WEBSOM and the system can efficiently process over a million documents [27]. Thus, the techniques used in WEBSOM can provide viable options for processing the enormous amount of content in documents from the web.

The authors have also observed that multiplying the vector of (encoded) words by a random matrix that produces a vector of a smaller dimension can reduce the vocabulary significantly while causing only a minor loss of information. Using these techniques, the dimension of the document vectors can be reduced to hundreds, at which point training the map using hundreds of thousands of documents is feasible.

6.2.3. Experiments – interpreting SOMs

We have mentioned that merely training a SOM is not of much value without a proper interpretation. While doing so, we still want to focus on our primary task throughout this work – sentiment analysis. In this section, we limit ourselves to observing whether the clusters created by SOMs correlate with the sentiment (in spite of being formed independently from it). If it were so, the sentiment of the majority of the instances in a certain cluster could indicate the sentiment of a new instance that falls into this cluster. This information could then be used as an additional factor in a classifier (which would probably be based on supervised learning). The following sections explore other applications of SOMs in combination with previously used techniques.

The dataset of 4671 tweets from SemEval 2013, Task 2, which was described and applied in Chapter 4, is used to train the SOM models presented in this chapter unless stated otherwise. The attribute containing the sentiment class (positive, neutral or negative) was excluded, leaving 1000 attributes of values either 0 or 1, indicating the presence of selected phrases in the tweets.

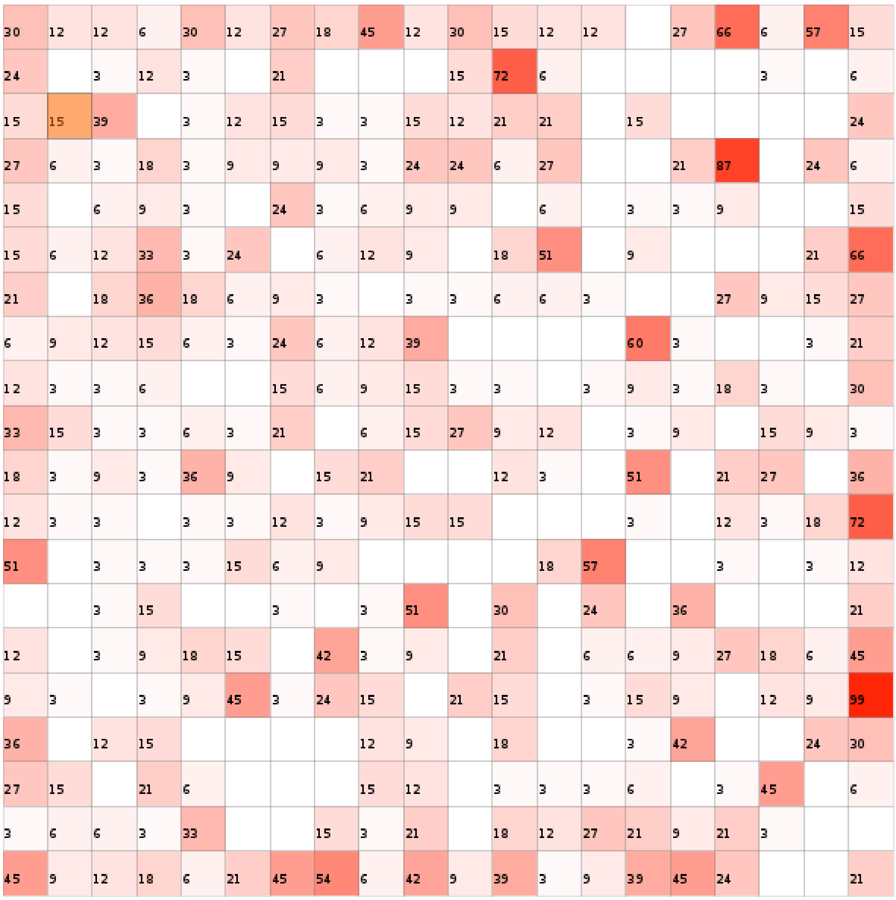

FIGURE 8: A SOM OF 20X20 NEURONS WITH A GRID TOPOLOGY TRAINED ON THE SEMEVAL 2013 DATASET. EACH NEURON IS CONNECTED TO ITS 4 NEIGHBOURS. THE MAP WAS TRAINED IN 15000 ITERATIONS.

When the clusters were formed, we wanted to find out whether some of them correspond, to one of the sentiment classes of the tweets. If it were so, the cluster assigned to a new tweet could serve as an additional information that might be used, e.g., by a hybrid classifier. First, we looked for clusters where most of the instances belonged to a single sentiment class. On a 16×16 SOM, similar to the one displayed in Figure 8, multiple neurons were found that scored up to 90 hits (the dataset had 1671 instances) and over 90% of the tweets contained had the same sentiment class.

We wanted to create a metric that would allow us to measure how much a neuron in the SOM is more or less correlated with sentiment. The metric was based on the assumption that a neuron that would be most correlated with one sentiment class would have weights that, for each attribute, mirror the correlation of the attribute with the given sentiment class. For every neuron N, we computed the similarity for each sentiment class S as

$$sim(N, S) = \sum_{i=1}^{n} w_{N,i} c(a_i, S) \tag{38}$$

where $c(a_i, S)$ is the correlation coefficient between the indicator variable that labels the tweets belonging to class S and the value of the attribute a_i. $w_{N,i}$ is the weight of the neuron corresponding to attribute a_i. For example, if there were three attributes corresponding to the stemmed words "thank", "sad" and "great" whose correlations to the positive sentiment class would be 0.5, -0.6 and 0.7 and a neuron with weights 0.9, 0.1 and 0.8 would score a similarity of 0.81.

We tested the performance of this metric on a smaller, 8×8 SOM. For each neuron N, we found the class S_{max} for which $sim(N, S_{max})$ was the highest and considered a classifier that would assign this class to all the instances for which N was the winning neuron. For the top 5 neurons (with the highest $sim(N, S_{max})$), which together contained 140 instances, this assumption was correct in over 85% cases. SOM clustering is unlikely to provide the basis for the correct classification of large portions of the dataset, as the clusters were not constructed based on the sentiment and the correlation can be explained as coincidental. This was confirmed by the relatively small size of the well-correlated clusters. On the other hand, if smaller clusters appear that indicate a certain class with a good level of confidence, we could use this information in combination with a supervised classifier.

6.2.4. Building a hierarchical SOM

The basic SOM model has the architecture of a flat grid. However, the data that is processed using SOMs, such as digital journal archives, library catalogues and web documents, often have a hierarchical structure. A hierarchically organized map is also easier to navigate from the user perspective. Moreover, the performance of learning algorithms can be increased by breaking the clustering problem into several smaller parts.

Hierarchical SOM models [28] consist of a set of SOM layers arranged so that higher levels perform a more general classification before sending the input vector to the lower-level SOM corresponding to the winning neuron of the higher level. The lower-level SOM then performs more detailed clustering and potentially passes the vector further down an appropriate path.

The disadvantage of a static hierarchical SOM is that the user needs to pre-set the network structure and thus make a reasonable prediction about the number of levels and size of the maps at each level that will be needed. This is not an easy task as the data could be unbalanced – more detailed clustering may be needed in some parts of the SOM tree than elsewhere. To address this issue, the so-called growing hierarchical SOM models have been developed. They begin with a simple small SOM (usually with a 2D-grid topology) and add columns, rows and layers as needed to fit the data [29].

As we wanted to test the performance of hierarchical SOMs on the SemEval dataset, we looked for an implementation of WEBSOM or a similar system based on hierarchical SOMs that could be freely used for evaluation. An implementation in the form of a Weka clustering function would have been ideal, as the most of the models presented in this work so far were created using Weka. However, we did find neither an opensource WEBSOM implementation, nor a hierarchical SOM suitable for direct use with Weka. After experimenting with various options, we discovered that fragments of a project that has been developed at the Vienna University of Technology can be integrated with Weka with surprising ease. Here we would briefly mention this project from the technical point of view.

The SOMLib Digital Library Project [30] is based on the idea to apply an organization similar to that of a library to vast digital archives and potentially the World Wide Web as a whole. Emphasis is placed on visualization and the ease of use of the final structures. The system provides a user interface that resembles a traditional library [31] not only by the clustering of topics (which is done via SOMs, similarly to WEBSOM), but also by automatic labelling along with other bibliographic information related to the documents.

The SOMLib core uses two main files to represent the datasets. A template vector file that defines the terms related to vectors in the input space of the SOM. The second file is a collection of the vector-like representations of the instances from the dataset [32]. This format is very similar to that of the Weka ARFF files, except that the Weka datasets are stored in a single file [33]. ARFF files begin with the declaration of attributes – words or n-grams, analogically to the template vector file of SOMLib. The second part of a standard ARFF file contains word vectors produced by appropriate filters, representing the instances of the dataset in terms of the attributes declared beforehand. By default, the StringToVector filter of Weka produces binary indicators of the occurrence of selected terms in the texts, But TF and IDF transformations are available. The SOMLib input vector file format accepts both binary and TF-IDF-transformed inputs when this is properly indicated in the $TYPE property in the header of the file.

Thus, the ARFF files can be transformed quite directly to SOMLib inputs. SOMLib can then produce either a classical SOM, a growing SOM (growing SOMs are discussed in Section 6.5), or a growing hierarchical SOM where the complexity and size of the map is governed by the required precision the model trains towards. The result is available as a set of HTML pages, each containing a table of clusters with links either to the descendant level of the hierarchical SOM or to the source documents on the leaf level. An example of this can be seen in Figure 6.

The SOMLib library also uses a tokenizer and a stemmer to transform the respective web documents to the input file formats described above. This means the interoperability between Weka and SOMLib via the interface of the input files can be also experimented with in the opposite direction, that is, using the preprocessing modules of SOMLib with the learning algorithms of Weka.

6.3. Separate models for clusters

SOM models presented in 6.2 revealed clusters that correlate with the sentiment of the tweets. Some of these clusters can be used to classify the sentiment of unknown tweets, albeit the precision is significantly worse than that of a trained supervised classifier. This led us to the idea of combining supervised and unsupervised learning by using a clustering algorithm to divide the dataset into subsets that have similar traits. Although these traits may not be directly related to sentiment, it may be easier for a supervised learning algorithm to build a model for each of the subsets due to a lesser diversity in the data.

We used the SemEval 2013, task 2 dataset with 4671 tweets, which had also been used in Chapter 4 to evaluate the performance of SVMs and to compare it to other models from previous chapters. Without specifying the learning algorithms used at this point, the sequence we used to determine the effectiveness of a combined model is as follows:

1. Extract the 10000 most frequent 1-3-grams from this dataset as attributes.
2. Divide the dataset into a training set containing $2/3$ of the data and a test set containing the remaining $\frac{1}{3}$.
3. Use the training set to select the 1000 best attributes based on their correlation with the target class. Restrict the set of attributes for both the training and test sets to these 1000 attributes.
4. Remove the target attribute (sentiment) from the training data and use the resulting set to train a clustering model using unsupervised learning.
5. Divide the training set (containing the sentiment attribute) into subsets using the clustering model. Perform the same division on the test dataset, using the clustering model trained on the training set.
6. Now we have a training set and a test set for each of the clusters. For each of the clusters, train a classifier using a supervised learning algorithm on the respective training set and evaluate its precision on the test set.
7. Train a model on the whole training set (not divided by clustering) using the same supervised algorithm as in step 6. Evaluate its precision using the whole test set (not divided by clustering).
8. Compare the average precision of models created in 6. and the precision of the model built in step 7.

For clustering, we used the SOMLib library discussed in 6.2.4. We did not want the SOM to have too many clusters as the instances of each cluster were going to be used for training of a supervised classifier that had to have a sufficient number of instances available. We neither wanted to set the structure of the SOM statically as we could not predict the ideal structure and also wanted the overall method to be as general as possible. Therefore, we used the growing hierarchical SOM method that continues adding new layers to the hierarchy and new neurons to the existing layers until the model fits the data with a given accuracy (the library uses the parameters τ_1 and τ_2 to set the target accuracy). After a few experiments, we set the parameters of the learner so that the top level of the SOM hierarchy would contain between 5 and 10 neurons and the overall learning process would take approximately 6 hours using a 8-core 2.4 GHz processor.

For the following supervised learning, we used the SVM model as described in Chapter 4, since it achieved a satisfactory performance in the experiments described in Section 4.4. We used the default settings of Weka, as the goal here was not to build the best classifier, but merely to compare the performance of the classifier with and without the clustering preprocessing step. This is also the reason why we reused the SemEval dataset which had been cleaned by the authors and higher classification accuracies can therefore be obtained than for raw data, as noted in Section 4.4.

We repeated the experiment two times, each time obtaining similar results. Here, we provide the results from the second experiment. The clustering phase produced a 3-level hierarchy with the top level node containing 6 clusters, depicted in Figure 9.

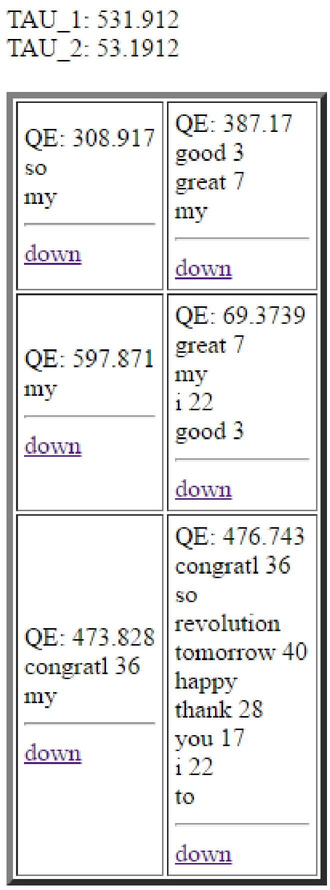

FIGURE 9: THE TOP LEVEL OF THE HIERARCHICAL SOM PRODUCED BY THE GHSOM ALGORITHM OF SOMLIB FOR THE TRAINING SET CONTAINING 3114 INSTANCES (NOT USED IN TESTING). FOR EACH CLUSTER, THE REPRESENTATIVE STEMMED N-GRAMS ARE SHOWN.

As stated above, for each of the clusters, a SVM model was trained using the respective training data and tested using the corresponding test set. The results are summarized in Table 7.

	Correctly classified instances	Incorrectly classified instances	Precision
Cluster 1	278	7	0.975
Cluster 2	322	17	0.950
Cluster 3	397	18	0.957
Cluster 4	6	0	1
Cluster 5	315	10	0.969
Cluster 6	182	5	0.973
Clusters total	**1500**	**57**	**0.963**
Single model	1434	123	0.921

TABLE 7: THE PRECISION OF SVM MODELS TRAINED SEPARATELY FOR EACH CLUSTER COMPARED TO THE PRECISION OF A SINGLE MODEL TRAINED USING THE WHOLE NON-CLUSTERED TRAINING SET. THE PRECISION WAS MEASURED USING THE TEST SET OF 1557 INSTANCES.

As the table shows, the models trained separately on the respective clusters produced by the SOM are consistently more precise than the model trained for the whole training set of 3114 instances and tested on the whole test set of 1557 instances. This may be, in part, due to the fact that multiple keywords associated with the clusters, as shown in Figure 9, are related to sentiment.

Since the training algorithm of the SOM was unaware of the sentiment attribute, it may be by chance that the clusters were constructed in a way that was suitable for sentiment analysis.

On the other hand, when training, it is not a flaw to train multiple models and use the one that better suits our needs. The SOM, although trained independently of sentiment, can be reviewed manually using the HTML output of the SOMLib library in order to see if the clusters formed are sentiment-related. Also, as stated, we performed the experiment two times with similar results.

In conclusion, the results of the experiments performed in this section indicate that clustering can improve the precision of a model built subsequently using supervised learning. Further experiments may be necessary in order to either challenge or confirm this assumption.

6.4. Clustering and tensor networks

After testing the effects of clustering on SVMs, we wanted to perform similar tests with neural tensor networks discussed in Chapter 3. This was more problematic than expected due to the fact that the training set provided for tensor networks by the authors uses a specific data format describing the parse trees and the sentiment of each node in each tree. The format we needed for SOM training required attributes indicating the presence of selected n-grams, which was also used in multiple previous chapters.

We extracted the text from the trees and transformed it into attributes in the same manner as the tweets in Chapter 4 and trained the SOM using this dataset. We then proceeded to split the new dataset based on the clusters of the SOM and map its instances back to the original TNN dataset, splitting it accordingly. We used the built-in training procedure of the Stanford NLP sentiment classifier to train a separate tensor network for each cluster.

From the Stanford Sentiment Treebank, 8544 short texts were used for training and another 1101 for testing, the volume of the data was therefore comparable to the dataset used to train SVMs. The results were less so. While the tensor model achieved over 78% precision on the test set when the whole training set was used for training, after the dataset was split, the best-performing clusters achieved only 67% precision on their corresponding test sets and most were only trained to a precision of $30 - 40\%$.

To explain why the performance of the tensor model was lowered by clustering, while that of SVMs was slightly improved, we could compare the complexity of both models. SVM models, whether trained for a single cluster or the whole dataset, were trained in mere seconds, compared to several hours in the case of TNNs. More complex models will probably need more time and data to be trained properly, that is for the value of the error function they minimize to stabilize.

The format difference may also have played a role – the clusters were trained using a different representation of the data than the per-cluster TNNs. This was, however, dictated by necessity as the SOMLib GHSOM library only accepts input in the form of attribute vectors and TNNs, naturally, require the node-level sentiment classification to be provided in the parse trees.

6.5. Growing neural gas networks

Clustering algorithms build models that divide the input space into areas, ideally small and densely populated by the training instances. This division can be represented by a set of boundaries or by a limited number of clusters. The so-called growing neural gas (GNG) networks also solve the problem of covering the data by clusters or, more precisely, neurons. GNGs, however, are more influenced by the biology-inspired approach to unsupervised learning known as competitive Hebbian learning [34].

Hebbian learning is based on the observation of the brain where cells that are often stimulated by the same signal form a minimal interconnection that is strengthened as the interaction continues. This is to make the interaction more efficient in the future.

[35] used the ideas of Hebbian learning, as well as those of Kohonen maps, to propose a new method for solving the problem of learning the topology of the data. The topology is represented by a neural network where each neuron is assigned a vector representing the position of the neuron in the input space. The input instances are iterated repeatedly and the closest neurons are moved slightly towards them at a diminishing pace. This is apparently very similar to the principle of Kohonen networks when we omit the topological connections between neighbouring neurons.

What sets the Fritzke's model apart is the ability to create new neurons. In fact, the model also places edges between neurons, but these are not used for direct interaction between the neurons like in Kohonen networks, but rather to reflect the changing topology of the network. When the network is presented with an instance $\boldsymbol{x}$, the closest neuron is located and the Euclidian distance between the weight vector $\boldsymbol{w}$ of the neuron and $\boldsymbol{x}$ is added to the neuron's cumulative error. After passing through the dataset, we obtain the error of each neuron. If there are two adjacent neurons with large errors, we can assume that there is a high concentration of instances between them. And if this error remains, the neurons have been unable to move towards these instances, because they are bound by other instances that keep them in their place. In this situation, a new neuron is created and placed between the two neurons that can represent these instances and edges are added to connect the new neuron to the two preceding ones.

Fritzke's method

0. Start with two neurons a and b at random positions $\boldsymbol{w}_a$ and $\boldsymbol{w}_b$.
1. Choose an input $\boldsymbol{x}$.
2. Find the nearest neuron s_1 and s_2.
3. Increment the age of all edges emanating from s_1.
4. Add the squared distance between the input $\boldsymbol{x}$ and the nearest neuron in the input space to a local counter variable: $\Delta error(s_1) = \left\|\boldsymbol{w}_{s_1} - \boldsymbol{x}\right\|^2$
5. Move s_1 and its direct topological neighbors towards $\boldsymbol{x}$ by fractions ϵ_b and ϵ_n, respectively, of the total difference vector:
   $\Delta\boldsymbol{w}_{s_1} = \epsilon_b(\boldsymbol{x} - \boldsymbol{w}_{s_1})$
   $\Delta\boldsymbol{w}_{\boldsymbol{n}} = \epsilon_n(\boldsymbol{x} - \boldsymbol{w}_n)$ for all direct neighbours n of s_1
6. If s_1 and s_2 are connected by an edge, set the age of this edge to zero. If such an edge does not exist, create it.
7. Remove all edges with an age larger than a_{max}. If this results in points having no emanating edges, remove them as well.
8. If the number of input signals generated so far is a multiple of the parameter λ, insert a new neuron into the network as follows:
   - Determine the neuron q with the maximum accumulated error.
   - Insert a new neuron r halfway between q and its neighbor f with
   the largest error variable: $\boldsymbol{w}_r = 0.5(\boldsymbol{w}_q + \boldsymbol{w}_f)$.
   - Insert edges connecting the new unit r with units q and f, and remove the original edge between q and f. The age of these edges is initialized to 0
   - Decrease the error variables of q and f by multiplying them by the parameter α. Initialize the error variable of r with the new value of the error variable of q.
9. Decrease all error variables by multiplying them with a constant d.
10. If a stopping criterion (e.g., net size or some performance measure) is not yet fulfilled go to Step 1.

Complementarily, in terms of biology, connections between the neurons can "age" and be eventually lost when the neurons do not interact for a prolonged period of time. In the Fritzke's model this corresponds to removing the edge between two neurons if, in a given number of previous passes through the training set, no instances were found for which these two neurons would be the closest ones.

It is also worth noting that efficient semi-supervised learning methods exist based on the GNG [36]. These begin by training the network using labelled instances and then gradually add new unlabelled instances to the neural gas. The network labels the new instance according to the closest neuron. Afterwards, the instance is added to the set of labelled instances and the network is further trained using the whole labelled set. This has been shown to increase the performance significantly, especially when the labelled dataset is small and there is a large set of unlabelled data available. This is definitely the case with sentiment classification on social networks, where professionally labelled datasets usually contain only a few thousands of instances while tens of millions of messages are posted every day on Twitter. Therefore, semi-supervised neural gas networks can be a suitable approach to the task of sentiment analysis. This is, however, beyond the scope of clustering, which we concern ourselves with in this chapter.

7. Convolutional neural networks

Convolutional networks are a special type of neural networks, purposely designed for image recognition in OCR engines, face recognition and other image recognition systems. The problem with image recognition is that while a higher quality of the images being classified generally leads to more precise models, learning these models can easily become computationally infeasible. For example, a relatively low-resolution picture of 100×100 pixels produces an input vector of 10000 pixels, which, in the case of colourful images, has to be further decomposed to the three colour channels, red green and blue, in order for the colour to be properly represented by three numbers in the range $[0,1]$. At the same time, 100×100 pixels can be insufficient for a photograph from which a face is to be reliably recognized.

In order to address this problem, convolutional neural networks take advantage of the "locality" of images. That is, images usually consist of regions with similar colour and a shape that is present in one part of the picture is likely to continue through the image. For example, when recognizing black letters on a white background, most of each image will be white and we need to detect the shape of the edges rather than give each pixel equal importance.

7.1. Architecture

The architecture of CNNs (well explained in [37]) aims to reduce the number of connections between neurons, thereby lowering the number of parameters to be trained and lifting the computational burden high-resolution images bring. This is achieved by dividing the image into windows, or patches, of smaller size. Fully connected neural networks are then formed only over aggregations of these windows, as we will see later. For a black and white image of $n \times n$ pixels (where we do not have to further decompose the pixels into multiple neuron inputs) and the window size of $m \times m, m < n$, we can cover the picture by $n - m + 1 \times n - m + 1$ windows, each time moving the frame by one pixel. The overlapping of the windows is advantageous – it allows the CNN to be translation invariant, meaning that the network will correctly identify two images as belonging to the same class, even in one of the images, the object captured is slightly shifted to one side, as is often the case with photographs and letters.

Using these windows, we need to capture the necessary features of the image. For example, in order to identify the letter A, we need to locate each of the three lines and their connections. A convolutional network therefore contains k feature maps which compute indicators of specific features of the image. Each of the k feature maps has to capture a different aspect of the image and therefore we want to "view it from a different perspective". More formally, a filter $\boldsymbol{x} \in \mathbb{R}^{m \times m}$ is applied to the $n - m + 1 \times n - m + 1$ overlapping windows to produce a feature map $\boldsymbol{c} \in \mathbb{R}^{n-m+1 \times n-m+1}$ [38]. Each window with its input vector $\boldsymbol{x} \in \mathbb{R}^{m \times m}$ is transformed to a single number c as follows:

$$c = f(\boldsymbol{wx} + b) \tag{39}$$

where b stands for the bias term and f is a non-linear function such as $tanh$. By this, we have already reduced the dimension of the input significantly. But we can reduce it further based on the local stability of images. Since the windows are only one pixel apart, we likely captured the same feature of the object in multiple windows. One of these inputs is sufficient to represent the feature, typically the strongest one, being the one in which the feature is captured most clearly. This technique is known as pooling. In practice, pooling includes dividing the image, now represented by the feature map, into small disjoint regions, and aggregating the features in the regions, usually by average or maximum. The size of the pools in most cases varies between 2×2 and 5×5 features [37].

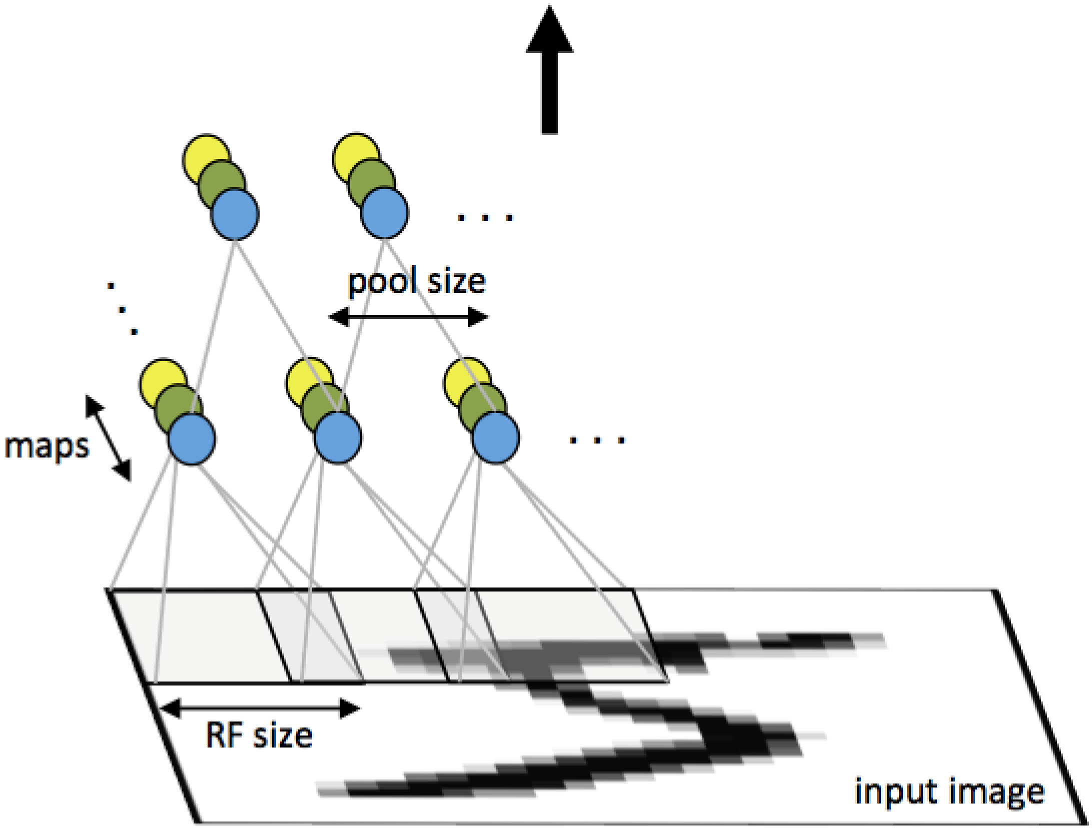

FIGURE 10: A CONVOLUTIONAL LAYER OF A NEURAL NETWORK [37]. THE COLOURFUL CIRCLES REPRESENT DIFFERENT FEATURE MAPS.

It is only after the pooling has been applied that a fully connected neural network layer is used over the resulting, transformed input. This whole sequence is regarded as one convolutional layer of a neural network (typically the first one). This can be followed by a number of fully connected layers or further convolutional layers.

7.2. Performance of text classification

By convolutional and subsampling layers, neural networks reduce the computational complexity that often rises when standard neural network models with densely connected neurons, such as multilayer perceptron, are used. At the same time, convolutional networks strive to achieve performance comparable to that of a fully connected network. In Section 4.4, the multilayer perceptron is mentioned briefly and it is noted that while performing worse than for example SVMs when given comparable amount of computation time, multilayer perceptrons could perform better when a larger network was used. To test this, the SemEval 2013, task 2 training dataset, used in Chapter 6, was used again to train a perceptron with 1000 input neurons (where the inputs indicate the presence of 1-3 – grams in a tweet) and 1 hidden layer of 100 neurons was trained in 1000 iterations. Training a single model took over 2 hours of a 6-core processor time, meaning 10-fold cross-validation took close to 24 hours. Nevertheless, the average precision of the model was 95.6326 %.

This was compared to a convolutional network using the same input data. Although convolutional networks are not primarily designed for text classification and the order of input neurons did not reflect the order of the words in a tweet, we wanted to test whether convolutions can be a viable method of dimensionality reduction in this case. Also, the input neurons were ordered by the correlation of the n-grams with sentiment, which means that a simultaneous activation of neighbouring neurons, especially on the "more significant" end of the input vector, is a notable factor that the convolution technique can help capture. After initial experiments, a network with 20 feature maps, window size 25 and pool size 4 was used. A single model was trained in 310 seconds, as opposed to 2 hours in the case of a fully connected network. The average precision of the model was slightly lower – 91.159 %, which is close to the performance of the basic SVM model reported in 4.4.

The number of feature maps used showed to have the largest influence on the performance of the model. 20 feature maps were used, because there were not significant precision gains when using more feature maps. The results of these experiments are summarized in Table 8. The table indicates that the factors influencing the sentiment of a tweet can probably be represented by 20 suitably chosen attributes.

# Feature maps	5	10	15	20	25	30
Training time (s)	67	187	338	310	1263	906
Precision	0.760	0.850	0.858	0.912	0.905	0.919

TABLE 8: COMPARISON OF THE PERFORMANCE OF A CONVOLUTIONAL NEURAL NETWORK USING THE WINDOW SIZE OF 25 NEURONS AND POOL SIZE OF 4 WITH VARIOUS NUMBERS OF FEATURE MAPS. THE TRAINING TIME IS NOT EXACT, AS THE SERVER WAS ALSO PERFORMING OTHER TASKS AT THE TIME, BUT IS A SUFFICIENT INDICATOR OF THE TREND.

Interestingly, adding multiple CNN layers did not improve the performance. In some cases, it even led to a slight deterioration, while the training of the model took approximately 2 times as long.

These results show that convolutional networks can perform comparably well to fully connected models. Fully connected networks are still likely to achieve a higher precision when trained thoroughly, but the time required for their training is significantly higher. The speed of training is an important factor, especially in the context of neural networks and other data originating from the internet, where vast datasets are often encountered and simpler models can be more successful since a larger portion of the data can be processed.

7.3. Convolutional networks and clustering

Since convolutional networks performed comparably well to SVMs discussed in Chapter 4, we tested whether clustering can have a similar effect to that covered in Section 6.3. There, the average precision of SVM models was increased by 4% when the input data was divided into 6 subsets using a SOM and a separate model was trained and tested for each of these subsets as opposed to a single model created using all the training data.

	Correctly classified instances	Incorrectly classified instances	Precision
Cluster 1	281	4	0.986
Cluster 2	319	20	0.941
Cluster 3	402	13	0.969
Cluster 4	6	0	1
Cluster 5	323	2	0.994
Cluster 6	186	1	0.995
Clusters total	**1512**	**40**	**0.974**
Single model	1442	115	0.926

TABLE 9: THE PRECISION OF CONVOLUTIONAL MODELS TRAINED SEPARATELY FOR EACH CLUSTER COMPARED TO THE PRECISION OF A SINGLE MODEL TRAINED USING THE WHOLE NON-CLUSTERED TRAINING SET. THE PRECISION WAS MEASURED USING THE TEST SET OF 1557 TEST INSTANCES (NOT USED IN TRAINING). COMPARE THIS TO TABLE 7.

Here, we used the same hierarchical SOM model and trained a convolutional network with 20 feature maps for each of the clusters on $\frac{2}{3}$ of the data. The models were then tested using the remaining $\frac{1}{3}$ of the dataset. The average results were compared to the results of a single convolutional network trained on the whole training dataset. The results are summarized in Table 9. The experiment was repeated 3 times with similar results.

The effect of clustering on the performance of convolutional networks is apparently similar to the case of SVMs. It is still possible that the clustering results were flawed, on the other hand, during the experiments with SVMs in Section 6.3, two separate SOMs were trained as well and the clusters of each were used to train a set of SVMs with comparable results. As mentioned already in Section 6.3, the cleaned SemEval training dataset itself may not be an accurate representation of real-world data, but it suffices if we only want to compare the performance of different algorithms.

8. Test application - SentimentNN

The majority of tests performed in this work used the Weka machine learning framework. The origin of the data processed and the key aspects of the setup of each experiment are mentioned in the respective sections. Still, this does not provide an end-to-end demonstration of the functionality. Also, repeating the experiments would require some preparations including the acquiry of data from Twitter, installation of specific Weka packages and the setup of the GHSOM training library (see 6.2.4).

To make this cumbersome process easier for the user, an application was created that performs the installation of Weka and necessary extensions. After the installation, this application can be used to perform the discussed experiments, particularly using the SVM and convolutional NN models. These experiments also comprise the entire testing process from the original Twitter data provided by the SemEval competition. The usage of SOMs is also demonstrated, although the training of hierarchical SOMs is not included. The reason is that the SOMLib implementation used in this work was built for the UNIX operating system, while the majority of the experiments were performed on a Windows machine, using custom code in .NET. Because of this, the test application covered in this chapter is also built for Windows systems. Therefore, there is one pre-trained SOM provided. The user can also train their own SOMs using a Linux machine, the GHSOM binary included on the CD and their own training configurations. This test application assists with the creation of SOMLib training inputs from Weka ARFF files, as described in 8.1.4.

8.1. Usage and experiment details

You can find the installation binaries of the application on the attached CD as “SentimentNN_Setup.exe”. The source code is available in the “SentimentNN” directory. The SentimentNN_Setup.exe contains the application itself, alongside with the test data and the installation of Weka 3.8 (which includes the installation of the Java Runtime Environment).

8.1.1. Installation

It is recommended that the target machine is running Windows 7 or higher or Windows Server 2012 or higher and has at least 8 GB of memory.

After launching the SentimentNN_Setup.exe program, you will be greeted by the screen shown in Figure 11. The following screen will prompt you to choose the installation folder. The executable of the application will be located in the “bin” subfolder of this folder and will be named “SentimentNN.exe”. This exe file will be launched after you finish the installation (unless you choose otherwise). After the installer has copied the files, the Weka 3.8 installer will be launched automatically, as shown in Figure 12. If you already have Weka 3.7 or 3.8 or a higher version installed, you may want to cancel this installation. When installing Weka 3.8, the Java Runtime Environment should be also installed, if it has not been installed already. If you encounter problems launching the test application, missing Weka or Java installation may be the cause.

After the setup finishes, the application is not yet fully ready for use. The NeuralNetwork package probably still needs to be installed. This is the first thing the application does when launched. The commands called and their output will be displayed in the output buffer in the central part of the window, as shown in Figure 13. The application checks whether NeuralNetwork is already installed and if not, installs it from Initialization\NeuralNetwork_1.1.zip in your installation directory. If you encounter problems during this phase, you can install the package manually from the GUI of the Weka package manager. After you install the package manually, start the application again. You should see a list of installed packages containing the NeuralNetwork package. Afterwards, the message “Initialization finished successfully” will be written to the output buffer, indicating that the application is ready for use. This means that both the Weka installation and the NeuralNetwork package have been found, the paths to Weka and Java have been added to the configuration file in bin\SentimentNN.exe.config and the “configured” switch in this file has been set to “true”. If you encounter severe problems, you may want to finish the installation of Java, Weka and NeuralNetwork yourself and set the appropriate values in the configuration file. After a successful configuration, the file should be similar to:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
<startup>
        <supportedRuntime version="v4.0"
sku=".NETFramework,Version=v4.5.2" />
</startup>
<appSettings>
<add key="configured" value="true" />
<add key="wekaPath" value="C:\Program Files\Weka-3-7" />
<add key="javaPath" value="C:\Program Files
(x86)\Java\jre1.8.0_65\bin\java.exe" />
</appSettings>
</configuration>
```

Once you have finished the initialization phase, the application will not perform the initial checks when launched again.

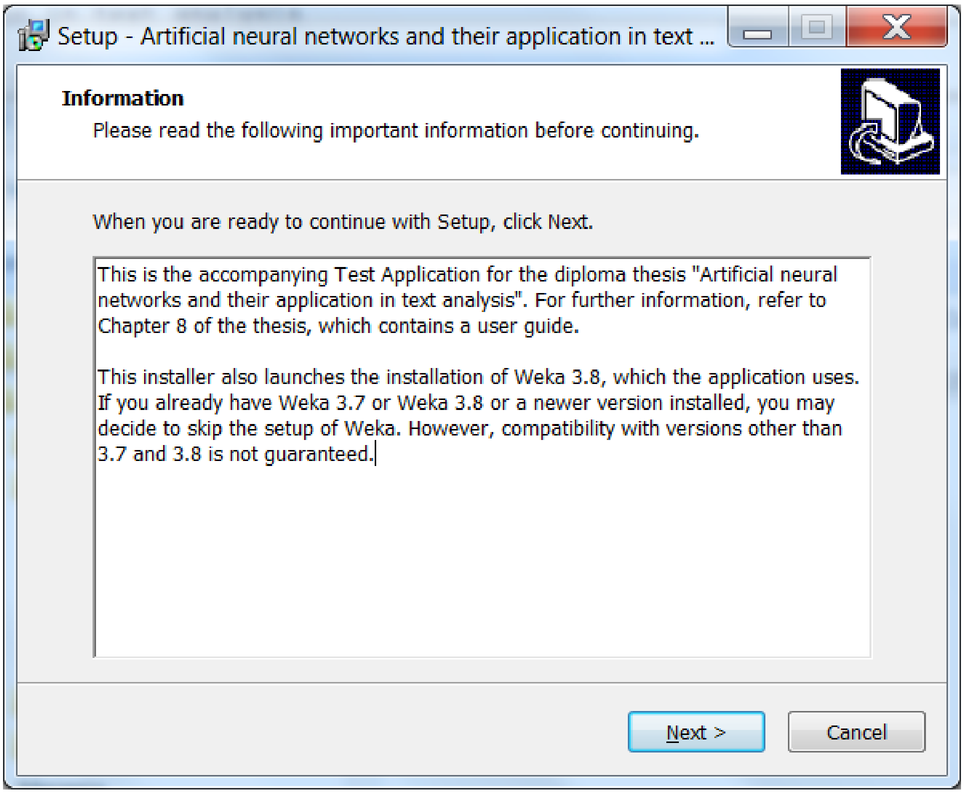

FIGURE 11: THE WELCOME SCREEN OF THE SETUP

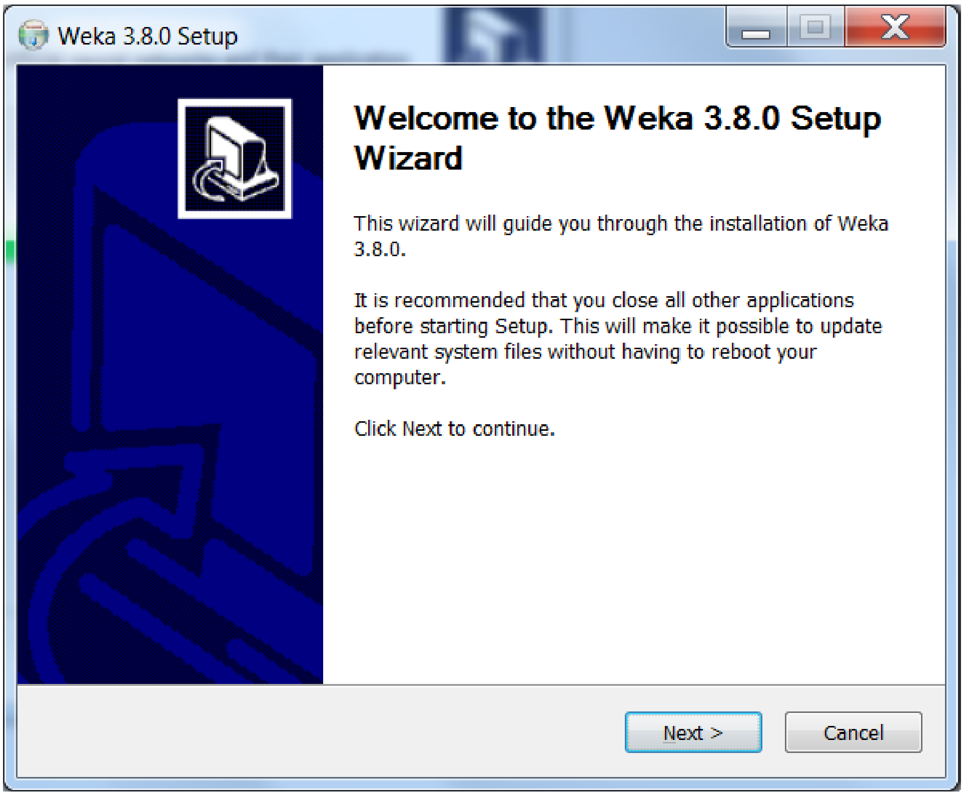

FIGURE 12: THE WEKA INSTALLER STARTS AUTOMATICALLY AT THE END OF THE SETUP. YOU MAY WANT TO CANCEL THIS IF YOU ALREADY HAVE A RECENT VERSION OF WEKA INSTALLED.

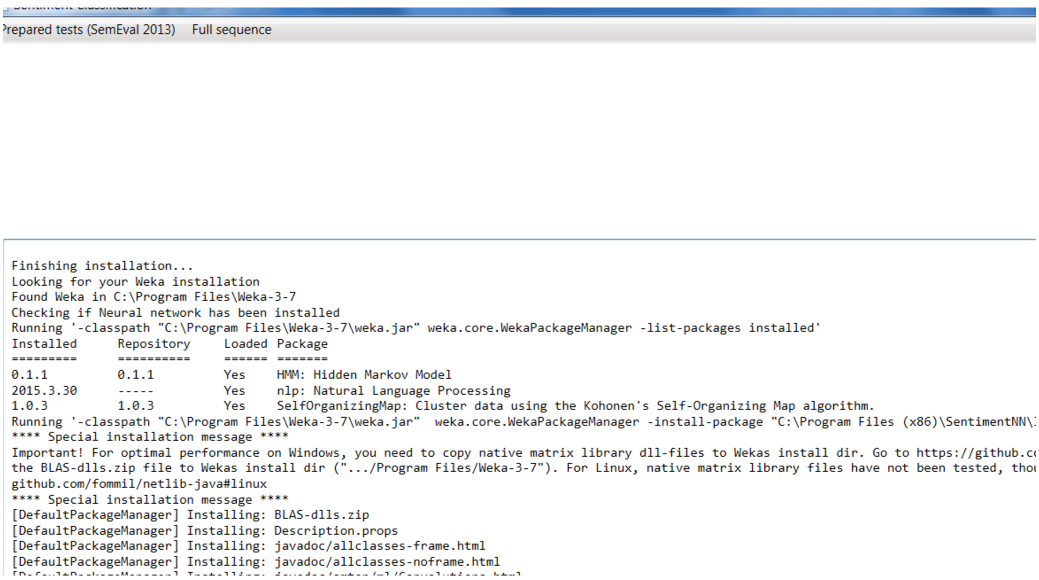

FIGURE 13: THE INITIALIZATION OF THE APPLICATION. THE APPLICATION SEARCHES FOR AN INSTALLATION OF THE NEURALNETWORK PACKAGE. IF IT IS NOT FOUND, IT WILL BE INSTALLED FROM THE INITIALIZATION\NEURALNETWORK_1.1.ZIP FILE IN YOUR INSTALLATION DIRECTORY. IF YOU ENCOUNTER PROBLEMS DURING THIS PHASE, YOU CAN INSTALL THE PACKAGE MANUALLY FROM THE GUI OF THE WEKA PACKAGE MANAGER.

8.1.2. Using prepared test sets

All the test data used by the application are located in the TestData subfolder of your installation directory. When a training model is launched, a new output buffer is created in the application to which the training process will write its output. You can switch between these output buffers using the panel in the top right corner of the application. When you run multiple tests, you may want to use the “Clear finished tests” button to remove output buffers of the processes that have already exited from the output buffer list.

The processes can be launched from the menu on the top of the window. In this section, we will explain the items in the first menu, named “Prepared tests (SemEval 2013)”. In fact, this data was used in tests discussed in multiple previous chapters.

For the prepared test sets, the Twitter data has been cleaned, vectorised and split into a test set and a training set. We will discuss the preprocessing procedure in more detail in Section 8.1.3, where you can actually run the procedure and observe the intermediate results of the transformations. The resulting test and training sets both consist of patterns comprising 1000 attributes that indicate the presence or absence of the respective $1-3$-grams in the tweets. The target attribute has one of three possible values (positive, neutral, negative) for each pattern.

The first two options (SVM and Convolutional NN) run the training and test of SVM and a convolutional network as discussed in Chapter 4 and 7, respectively. For SVMs, the Sequential Minimal Optimization (see 4.3) training algorithm is used with default configuration. This model serves mostly as a reference point when comparing the performance of other methods. The parameters actually applied are shown at the top of the output buffer and their exact meaning can be found at http://weka.sourceforge.net/ [42]. The output of the SVM classifier is shown in Figure 14.

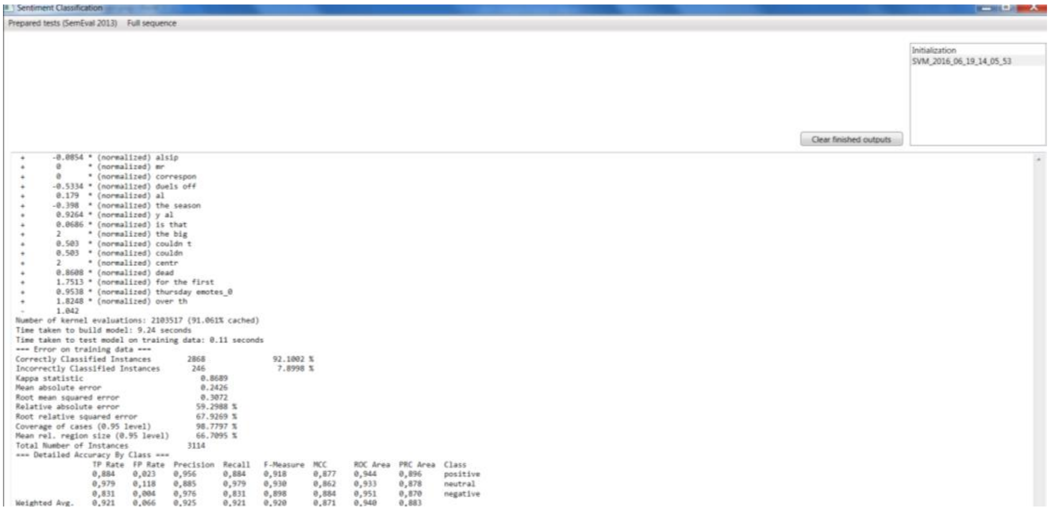

FIGURE 14: A TEST OF SVM FROM THE PREPARED DATASET. YOU CAN LAUNCH THIS TEST FROM PREPARED TESTS (SEMEVAL 2013) -> SVM.

In the case of convolutional networks, the configuration uses a window of 25 attributes, the pool size 4 and creates 20 feature maps. The network consists of only one convolutional layer, as adding multiple layers did not improve performance significantly in previous experiments (see 7.2). The maximum number of iterations is set to 1000, but the value of the error function usually stabilizes after no more than 100 iterations, at which point training will stop automatically.

The last two items in the “Prepared tests” menu use the same algorithms and configurations as the first two, but the dataset in the later options is split based on clusters formed by a hierarchical SOM trained using the SOMLib library as discussed in Chapter 6. As mentioned earlier, the training of hierarchical SOMs would be difficult to include in the test application, since the SOMLib library is built for UNIX systems and uses specific libraries that prevent it from running in a 64-bit environment even when used on a UNIX machine.

Nevertheless, Section 8.1.4 describes how to set up and train GHSOMs on a Linux machine with the assistance of the test application, which prepares the training data and can use the model trained by the user to split the dataset and perform an experiment analogous to the prepared one. Also, the outputs of the training of the SOM used to split the dataset in the “Prepared tests” is provided in the TestData\GHSOM subfolder of the installation folder, including HTML files (ifs_abstr_ghsom1_1_1_0_0.html contains the top level of the hierarchy), weight vectors (*.wgt files) and other descriptors used during the training of the SOM.

When you launch one of the two tests that use the SOM split, 6 output buffers are initialized at once as 6 independent training processes start. As the execution is parallelized, you can still launch other commands while the respective models are trained. Still, you may expect your machine to be busy for up to a few minutes.

8.1.3. Preprocessing

In the second part of the menu, labelled “Full sequence”, you have to run the “Prepare” command before any model can be trained. This transforms the original Twitter data saved in TestData\PrepSequence\orig.arff in a series of steps. These steps include (in this order)

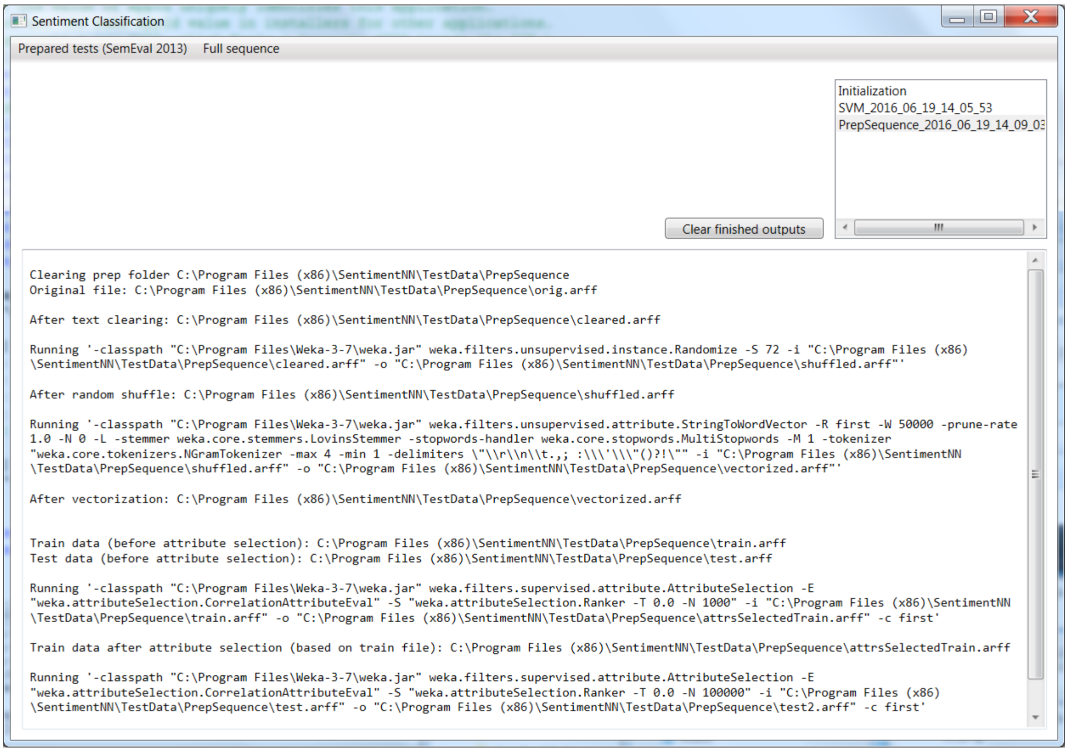

FIGURE 15: THE PREPARATION SEQUENCE LAUNCHED FROM FULL SEQUENCE -> PREPARE. IT STARTS FROM THE TESTDATA\PREPSEQUENCE\ORIG.ARFF FILE IN YOUR INSTALLATION DIRECTORY, WHICH IS THEN PASSED THROUGH STANDARD AND CUSTOM TRANSFORMATIONS. AFTER EACH TRANSFORMATION, THE PATH TO THE INPUT AND OUTPUT FILES IS WRITTEN TO THE OUTPUT BUFFER SO YOU CAN BROWSE THE INTERMEDIATE RESULTS AT WILL.

- Text clearing – regular expressions are used to remove http links from the text and replace user references (@-notations) with the neutral name “John”, which is not likely to be selected as an attribute
- Random shuffling of the tweets
- Vectorization – around 6000 $1-4$-grams will form the attributes of the dataset
- Splitting the dataset – $\frac{2}{3}$ of the data will be used for training and $\frac{1}{3}$ for testing
- Attribute selection – the training set is restricted to the 1000 attributes with the highest correlation with the target class

- Attribute synchronization – a custom .NET component is used to restrict the attributes of the training set to those that were selected for the training set in the previous step.

The preprocessing can take several seconds during which the application will be unresponsive.

The intermediate results are saved to the same directory. The resulting datasets are saved to attrsSelectedTrain.arff and attrsSelectedTest.arff. These files can be inspected manually or loaded in Weka (by the "Open file…" option on the "Preprocess" screen). By doing so, you can conduct arbitrary test within the capabilities of your Weka installations.

The run of the preprocessing sequence is displayed in Figure 15. After the preprocessing has been completed, the two lower options in the "Full sequence" menu are enabled. Figure 16 shows a convolutional neural network trained after the preparation. It also demonstrates how the user can stop the training before the stopping criterion is met in case the progress is slow.

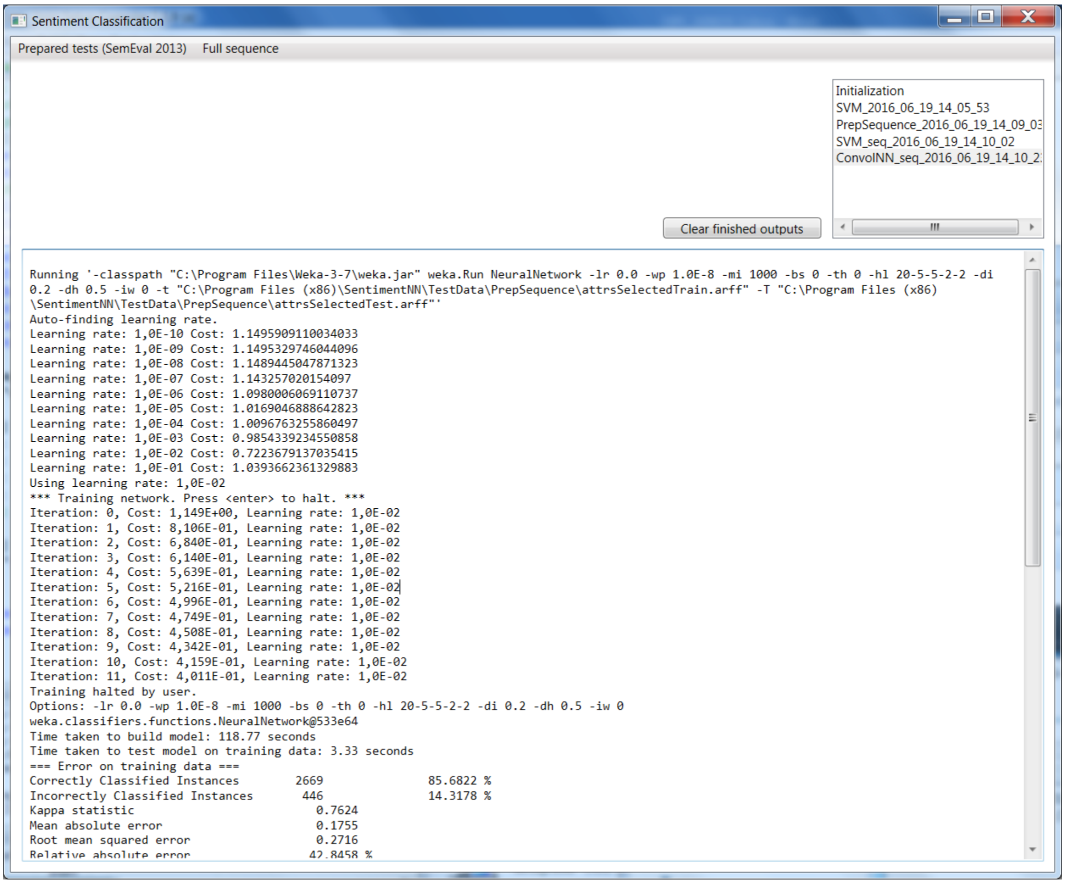

FIGURE 16: A CONVOLUTIONAL NN THAT WAS TRAINED USING FULL SEQUENCE -> CONVOLUTIONAL NN FROM THE MENU (AFTER THE PREPARATION SEQUENCE). THE MAXIMUM NUMBER OF ITERATIONS WAS SET TO 1000 (`-MI 1000`), BUT THE USER PRESSED ENTER AFTER 10 ITERATIONS. THE CURRENTLY RUNNING ITERATION WAS COMPLETED, AFTER WHICH TRAINING WAS TERMINATED AND A TEST WAS PERFORMED, AS CAN BE SEEN IN THE OUTPUT BUFFER.

8.1.4. Training a custom SOM

For training a custom SOM, you will need to run the GHSOM program included on the enclosed CD. This can be accomplished using a 32-bit Linux machine, as the library uses older versions of system libraries and it may not work on a 64-bit system. A virtual Linux machine can be downloaded from www.osboxes.org. You may also need to install standard C++ 5 libraries which may not be included in your Linux distribution.

Before training a SOM, the dataset needs to be prepared. The first step is preprocessing, described in 8.1.3, after which the "GHSOM" option in the "Full sequence" menu will be enabled. First, click on "Export for trainer" in the GHSOM submenu. This uses the training dataset in ARFF format which was created during the preprocessing phase to create two files in the TestData\GHSOM\train directory of your installation. The ghsomTrainSet.tv file describes the attribute vectors, while the ghsomTrainSet.in file contains the training data. You will also need a properties file, which configures the GHSOM trainer. A sample properties file is provided in the TestData\GHSOM\sampleProperties\sample.prop. The parameters used in the file are described at [43]. The most important parameters are *TAU_1* and *TAU_2*, which regulate the size and training time of the GHSOM. The *inputFile*, *descriptionFile* and *savePath* parameters need to be set properly to the location of the ghsomTrainSet.in, ghsomTrainSet.tv and the target output folder, respectively.

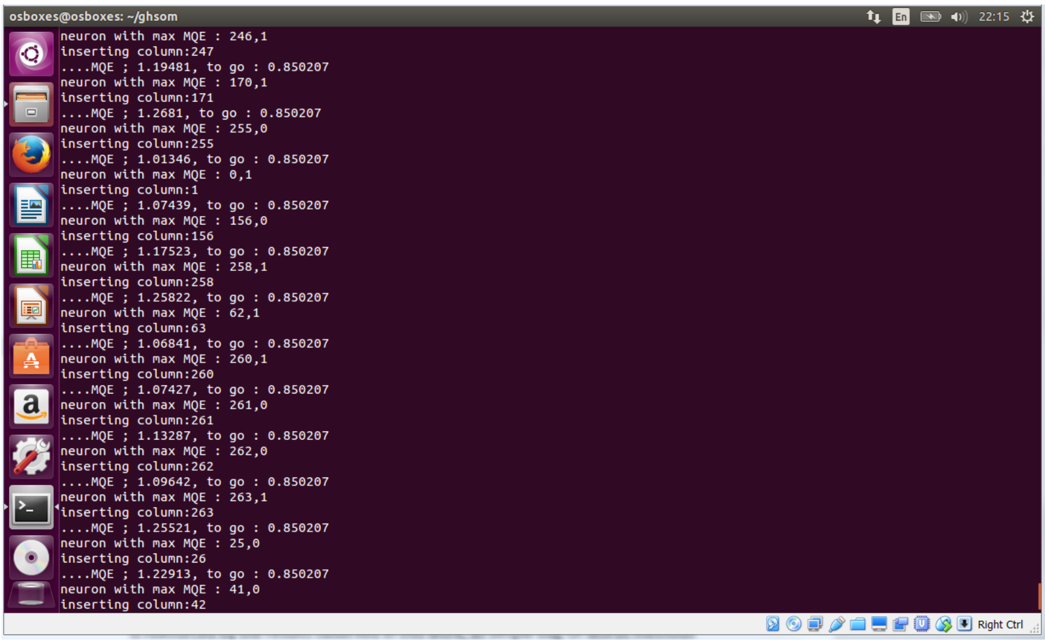

FIGURE 17: THE PROGRESS OF TRAINING A HIERARCHICAL SOM USING THE GHSOM LIBRARY.

After you have set the configuration, you can start the training the network by running `ghsom [propertyFile].prop`. The program will load the input and output the training progress, as shown in Figure 17. After it finishes, you will find the results in the configured *savePath* directory.

From the result hierarchical map, we will only be interested in the first level – this is stored in the first files in the input directory. [configuredPrefix]_1_1_0_0.html contains a html representation of the SOM with keywords. You can review these keywords to see whether the SOM is sentiment-related or whether it may need to be retrained. [configuredPrefix]_1_1_0_0.wgt contains the weight vectors of the neurons in the map. This is the file the test application will need in the next step. Copy this file to the TestData\GHSOM\weights folder and make sure this is the only file in the folder. After this, click on "Split dataset" in the GHSOM submenu. This will use the weights file to split both the train and test files created during the preprocessing phase. The split files will be saved to TestData\GHSOM\trainSpit and TestData\GHSOM\testSplit, respectively. A sample of the weights file is already provided in TestData\GHSOM\ifs_abstr_ghsom1_1_1_0_0.wgt, so you can actually click "Split dataset" without training a GHSOM of your own and then train a model of your choice using the dataset now split into 9 training and 9 test parts

After the train and test datasets have been split according to the newly trained SOM, click on either "Train SVM" or "Train CNN" in the GHSOM submenu to train a SVM or a convolutional network for each of the split training files and test it using the corresponding test file.

8.2. Implementation highlights

The key to building the application was the cooperation with the Weka engine. For this purpose, the commandline interface Weka Primer [39] was used. While this provided the basic functionality, we were still faced with a few challenging tasks during the development, as we attempted to provide a simple user-friendly interface that does not require the user to interact when not necessary and allows the user to perform basic tests with minimal effort. On the other hand, we wanted to make both the data and exact configurations used by the application easily accessible so that they can be modified and used also in other applications if appropriate.

8.2.1. Custom packages

While the Weka Primer interface can perform most of the basic tasks like training and testing a model, it is by default limited to the standard set of Weka features which does not include custom packages of algorithms that can be installed through the GUI of the Weka package manager.

However, the user-installed packages can be used in a similar manner to the standard ones with the `Weka.Run` command [40]. This command searches for an installed class that matches or contains the name provided. If multiple matches are found, the user is prompted to choose one of them. For example, after installing the NeuralNetwork custom package [41] that includes the implementation of convolutional neural networks, a 10-fold crossvalidation can be performed by the following command:

```
C:\Program Files\Weka-3-8>java -classpath weka.jar weka.Run
NeuralNetwork -lr 0.0 -wp 1.0E-8 -mi 1000 -bs 0 -th 0 -hl
100 -di 0.2 -dh 0.5 –iw 0 -t "C:\data.arff" -x 10
```

The package manager itself is also accessible from the commandline. For example, one can check whether the NeuralNetwork package is installed by running the command

```
C:\Program Files\Weka-3-8>java -classpath weka.jar
weka.core.WekaPackageManager

-list-packages installed
```

And if the package is missing, it can be installed from the ZIP downloaded from the Johannes Amtén's GitHub page by

```
C:\Program Files\Weka-3-8>java -classpath weka.jar
weka.core.WekaPackageManager

-install-package "C:\NeuralNetwork_1.1.zip"
```

Indeed, these are the actions our application performs during initialization.

8.2.2. Interprocess communication

Probably the simplest form of the test application that would enable the user to perform the chosen experiments would be a console application with a strict sequence of commands. Even a batch script would be an option. However, in terms of user friendliness, a Windows, or more specifically, a WPF application using .NET 4.5, is more suitable. However, since the Primer is a command-based interface, this application needs to execute commandline commands and collect the results.

Also, it is suitable to facilitate parallel execution when possible. For example, the user may want to start all tests that use a particular dataset at the same time to compare their speed. In Weka 3.7, using default settings, a separate Weka window was needed for each algorithm one wished to run in parallel. This is not the case with the test application presented here, as most of the commands the user issues start a separate process (using the Java Runtime Environment) that is assigned a separate output buffer in the main application to which it redirects the output that would otherwise be sent to the command window (which is hidden).

In order to receive the output asynchronously handlers are created for the `System.Diagnostics.Process.OutputDataReceived` and `System.Diagnostics.Process.ErrorDataReceived` events. These handlers pass the newly received output to the main thread that manages the GUI of the window using the

`System.Threading.SynchronizationContext.Post` method. It is also important to call the `System.Diagnostics.Process.BeginOutputReadLine` and `System.Diagnostics.BeginErrorReadLine` methods before starting the process so that the application listens for messages from the redirected output stream.

On the other hand, not all processes are parallelized. For example, the preprocessing sequence that converts a set of input strings to attribute vectors, described in Section 8.1, must be performed in the given order and the following training of classifiers cannot begin before the preprocessing is finished. Thus, this operation blocks the GUI, forcing the user to wait for a few seconds.

Similarly to the output stream, the input stream of the Weka processes is redirected as well. In most cases, this would not be necessary and standard Weka algorithms do not receive user's input once the training has been started. However, Johannes Amtén in his implementation of the convolutional neural network enabled the user to stop the training interactively before the set stopping criterion is met by hitting Enter. This can be of advantage when you find that training progresses slower than anticipated and you cannot wait for the procedure to finish or when you see that the value of the error function has already stabilized and further training yields little improvement.

Therefore, after Enter is pressed while the user is currently viewing the output buffer of a running CNN training procedure, this key is written to the input stream of the corresponding process. This is the only case when user input is passed to a Weka process. For the purpose of sending a single keystroke to a background process, the older option based on the `System.Windows.Forms.SendKeys.Send` method was also considered, but writing an empty line to the redirected `System.Diagnostics.Process.StandardInput` stream showed to be simpler and more reliable.

Conclusion

This thesis is focused on sentiment classification of short text messages within social networks. This task has recently received the attention of numerous research projects, as social network data are becoming both easily available and of more value. Both the structure of social networks and the form of their data also evolve rapidly. This suggests that only minimal assumptions about the structure of the data should be made, if a widely applicable method is to be devised. This suggestion is reinforced by the phenomena observed in this work, as simple bag-of-words methods, such as SVMs in Chapter 4, performed better than more language-dependent ones. This also speaks in favor of the machine learning-based approach to sentiment analysis over the use of complex lexicon-based systems, although lexicon-based models are the current industry standard.

While making the minimal possible assumptions about the structure of the text, the models trained to represent these texts can be increasingly complex. In this work, in particular the SVMs and a method combining neural networks and clustering were tested with promising results. While these results need to be verified also for data of a different origin, they indicate that a divide and conquer approach can possibly lead to reliable systems encompassing a vast variety of types of data. Although such models might include larger structures, they would not necessarily be excessively complex if the same approach of covering each segment of the data separately could be used.

Most chapters of this thesis are focused on a single machine learning model. The summary at the end of each of these chapters includes observations of lesser and higher importance. Let us recapitulate the more important or frequent of these observations.

- Language-independent models performed better than models applying syntactic parsing, due to the frequent lack of proper syntactic structure in the data and due to the non-standard language used on the internet.
- Self-organized maps can be used to divide the data and train a separate model for each segment. In some cases (Chapters 6 and 7), this leads to better average performance of the models.
- Convolution can be used to reduce the computational complexity of neural networks for text mining, with accuracy comparable to that of a fully connected network.

Although exploring a wide variety of methods ranging from simple logistic regression to large neural networks, this work used a relatively small dataset. This does not adequately represent the full content of social networks, where millions of new entries are inserted daily.

The volume of the real data has several implications. The volume imposes the demand that the methods used to process this data be sufficiently fast as slower models may not be able to process on the incoming data in real time. On the other hand, it allows us to filter the data and to concern ourselves only with the portion of the data that meets a certain level of quality. It was e.g. observed that not all of the posts bearing negative sentiment contain a reasonable complaint, expressing the cause of the negative sentiment.

The sentiment class itself will in most cases be not the final product of a practical application. The product should contain useful information that can be acted upon. Therefore, we should restrain ourselves to those social network posts that are clearly related to a product, service or an aspect of the former ones. The posts then should be aggregated in order to present the results in an easily comprehensible manner. This should be one of the aims of focus of further research.

A. Contents of the CD

The enclosed CD contains these files folders:

- SentimentNN_Setup.exe – the installer of the test application described in Chapter 8
- SentimentNN – a folder containing the Visual Studio 2015 solution from which the application was built
- ghsomBinary – a folder containing the ghsom application from the SOMLib library for training GHSOMs, as described in 8.1.4.

References

[1] M. Hilbert and P. López, “The World's Technological Capacity to Store, Communicate, and Compute Information,” *Science,* pp. 60-65, 5 10 2011.

[2] J. Han, M. Kamber and J. Pei, Data Mining: Concepts and Techniques, Waltham, USA: Morgan Kaufmann Publishers, 2012.

[3] P. Chapman, J. Clinton, R. Kerber, T. Khabaza, T. Reinartz, C. Shearer and R. Wirth, “CRISP-DM 1.0,” SPSS, 2000.

[4] J. Pilmann, Data and data clustering, Prague: Charles university, 2010.

[5] H. Ottar, “Linear versus logistic regression,” *Springer Science+Business Media B.V.,* 2007.

[6] A. Go, R. Bhayani and L. Huang, “Sentiment140,” 8 10 2015. [Online]. Available: http://help.sentiment140.com/for-students.

[7] University of Waikato, “Class Logistic,” 9 1 2016. [Online]. Available: http://weka.sourceforge.net/doc.dev/weka/classifiers/functions/Logistic.html.

[8] L. Reinecke and S. Trepte, “Authenticity and well-being on social network sites: A two-wave longitudinal study on the effects of online authenticity and the positivity bias in SNS communication,” *Computers in Human Behavior,* pp. 95-102, 2013.

[9] S. l. H. J. C. v. Cessie, “Ridge Estimators in Logistic Regression,” *University of Leiden,* 1990.

[10] R. e. a. Socher, “Recursive Deep Models for Semantic Compositionality,” 2013.

[11] B. Liu, Sentiment Analysis: Mining Opinions, Sentiments and Emotions, Cambridge University Press, 2015.

[12] D. Klein and C. D. Manning, “Accurate Unlexicalized Parsing,” *ACL,* 2003.

[13] C. Goller and A. Kuchler, “Learning Task-Dependent Distributed Representations by,” *In Proc. of the ICNN,* 1996.

[14] P. e. a. Nakov, “SemEval-2016 Task 4: Sentiment Analysis in Twitter,” 21 11 2015. [Online]. Available: http://alt.qcri.org/semeval2016/task4/.

[15] Stanford University, “NLP Portal,” 26 6 2016. [Online]. Available: http://nlp.stanford.edu/sentiment/code.html.

[16] Stanford University, “Sentiment Analysis,” 21 11 2015. [Online]. Available: http://nlp.stanford.edu/sentiment/index.html.

[17] G. J. Gordon, “Support Vector Machines and,” 15 6 2004. [Online]. Available: https://www.cs.cmu.edu/~ggordon/SVMs/new-svms-and-kernels.pdf.

[18] V. Vapnik, Estimation of Dependences Based on Empirical Data, New York: Springer-Verlag, 1982.

[19] E. Osuna, R. Freund and F. Girosi, “An Improved Training Algorithm for Support Vector Machines,” in *IEEE NNSP*, 1997.

[20] W. Li, X. Zhang, C. Niu, Y. Jiang and R. Srihari, “An Expert Lexicon Approach to Identifying English Phrasal Verbs,” *ACL '03 Proceedings of the 41st Annual Meeting on Association for Computational Linguistics - Volume 1,* pp. 513-520, 2003.

[21] P. A. Machonis, “Look it up and Try it out: An Original Approach to Parsing Phrasal Verbs,” in *26th conference on Lexis and Grammar*, Bonifacio, 2007.

[22] A. Balahur, “Sentiment Analysis in Social Media Texts,” in *Proceedings of the 4th Workshop on Computational Approaches to Subjectivity, Sentiment and Social Media Analysis,*, Atlanta, Georgia, 2013.

[23] SentiStrength, 16 03 2016. [Online]. Available: http://sentistrength.wlv.ac.uk/.

[24] SlangIt, 16 3 2016. [Online]. Available: http://slangit.com/.

[25] S. P. Lloyd, “Least Squares Quantization in PCM,” in *IEEE Transactions on Information Theory, vol. IT-28, NO. 2*, 1982.

[26] X. Wu, CS 332: Data Mining, Burlington, Vermont 05405, USA, 2014.

[27] S. Kaski, T. Honkela, K. Lagus and T. Kohonen, “WEBSOM — Self-organizing maps of document collections,” in *Neurocomputing 21*, Helsinki, 1998.

[28] E. S. Chifu and I. A. Letia, “Text-Based Ontology Enrichment Using Hierarchical Self-Organizing Maps,” in *NatuReS-08*, Karlsruhe, Germany, 2008.

[29] E. Pampalk, G. Widmer and A. Chan, “A New Approach to Hierarchical Clustering and Structuring of Data with Self-Organizing Maps,” Technical University of Vienna, 2003.

[30] A. e. a. Rauber, “The SOMLib Digital Library,” 9 4 2016. [Online]. Available: http://www.ifs.tuwien.ac.at/~andi/somlib/.

[31] A. Rauber and H. Bina, “Visualizing electronic document repositories: Drawing books and papers in a digital library,” Department of Software Technlogy, Vienna University of Technlogy, Vienna, 2000.

[32] A. e. a. Rauber, “SOMLib Data Files,” 9 4 2016. [Online]. Available: http://www.ifs.tuwien.ac.at/~andi/somlib/download/SOMLib_Datafiles.html#history.

[33] University of Waikato, “ARFF,” 9 4 2016. [Online]. Available: http://weka.wikispaces.com/ARFF.

[34] C. Fyfe, Hebbian Learning and Negative Feedback Networks, London: Springer-Verlag, 2005.

[35] B. Fritzke, “Growing Neural Gas Network Learns Topologies,” *Advances in Neural Information Processing Systems 7,* 1995.

[36] S. M. Zaki and H. Yin, “A Semi-Supervised Learning Algorithm for Growing Neural Gas in Face Recognition,” Springer Science + Business Media B.V., 2008.

[37] Stanford University, “UFLDL Tutorial,” 21 5 2016. [Online]. Available: http://ufldl.stanford.edu/tutorial/supervised/.

[38] Y. Kim, “Convolutional Neural Networks for Sentence Classification,” New York University, New York, 2014.

[39] University of Waikato, “Primer,” 18 6 2016. [Online]. Available: https://weka.wikispaces.com/Primer.

[40] University of Waikato, “How do I use the package manager?,” 18 6 2016. [Online]. Available: https://weka.wikispaces.com/How+do+I+use+the+package+manager%3F.

[41] J. Amtén, “NeuralNetwork,” 21 05 2016. [Online]. Available: https://github.com/amten/NeuralNetwork.

[42] University of Waikato, “weka.sourceforge.net,” 19 6 2016. [Online]. Available: http://weka.sourceforge.net/doc.dev/weka/classifiers/functions/SMO.html.

[43] G. Rodríguez, “WWS 509 - Generalized Linear Statistical Models,” 3 10 2015. [Online]. Available: http://data.princeton.edu/wws509/.

[44] A. Ng, “Machine Learning,” 3 10 2015. [Online]. Available: https://www.coursera.org/learn/machine-learning.

[45] J. P. Hoffmann, Linear Regression Analysis: Assumptions and Applications, Birmingham: Brigham Young University , 2005.

[46] E. Gabasova, “Understanding social networks with F#,” New York, 2014.

[47] Apache Software Foundation, “OpenNLP,” 25 12 2015. [Online]. Available: https://opennlp.apache.org/.

[48] m. Hu and B. Liu, “Mining and summarizing customer reviews,” *Proceedings of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining,* 2004.

[49] L. Qiu, W. Zhang, C. Hu and H. Zhao, "SELC: A Self-Supervised Model for Sentiment Classification," *Proceedings of the 18th ACM Conference on Informaation and Knowledge Management,* 2009.

[50] K. Veselovska and A. Tamchyna, "Using Hand-crafted Rules in Aspect Based Sentiment Analysis," Charles University, Prague, 2014.

[51] Americal Physical Society, "Journal of Applied Physiology," 20 3 2016. [Online]. Available: http://jap.physiology.org/content/115/11/1714.

[52] T. Joachims, "Support Vector and Kernel Methods," 20 3 2016. [Online]. Available: http://www.cs.cornell.edu/courses/cs578/2003fa/slides_sigir03_tutorial-modified.v3.pdf.

[53] J. C. Platt, "Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines," Microsoft Research, Redmond, WA, 1998.

[54] T. Hastie and R. Tibshirani, "Classification by pairwise coupling," University of Toronto, Toronto, 1996.

[55] P. Rojas, Neural Networks - A Systematic Introduction, Berlin: Springer-Verlag, 1996.

[56] J. A. Bullinaria, "Self Organizing Maps: Fundamentals," 27 3 2016. [Online]. Available: http://www.cs.bham.ac.uk/~jxb/NN/l16.pdf.

[57] K. Lagus, T. Honkela, S. Kaski and T. Kohonen, Websom for Textual Data Mining, Springer-Verlag, 1998.

[58] T. Honkela, S. Kaski, K. Lagus and T. Kohonen, "Newsgroup Exploration with WEBSOM Method and Browsing Interface," Helsinki University of Technology, Otaniemi, 1996.

[59] C. N. d. Santos and M. Gatti, "Deep Convolutional Neural Networks for Sentiment Analysis of Short Texts," in *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers*, Dublin, Ireland, 2014.

[60] N. Kalchbrenner, E. Grefenstette and P. Blunsom, "A Convolutional Neural Network for Modelling Sentences," University of Oxford, Oxford, 2014.

[61] A. Johannes, "NeuralNetwork," 18 6 2016. [Online]. Available: https://github.com/amten/NeuralNetwork.