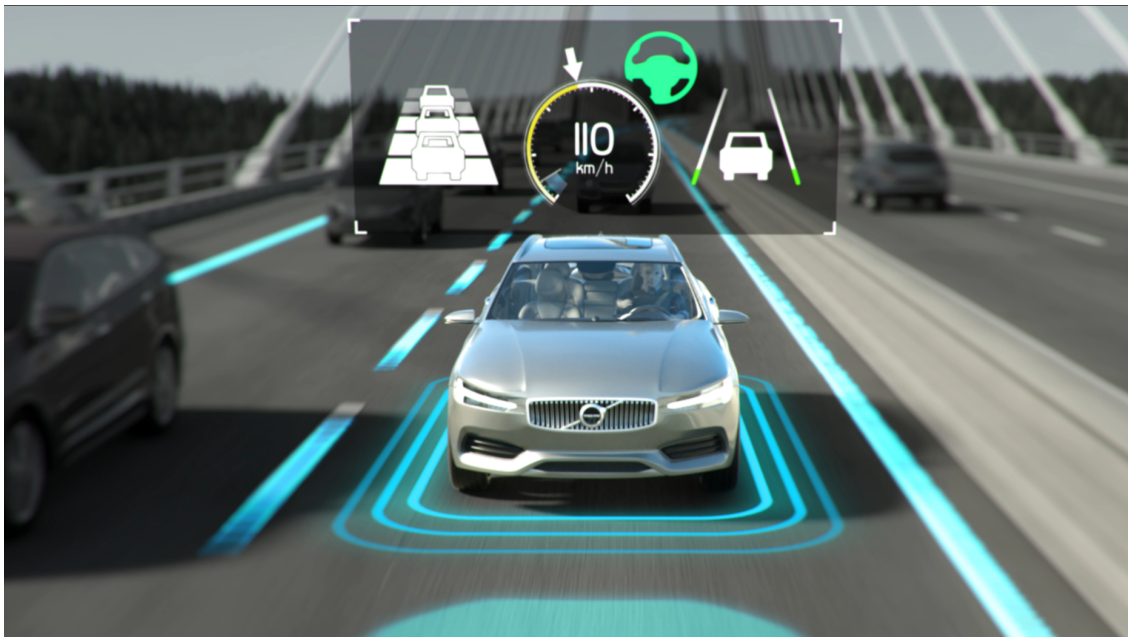


Multi-Class Text Classification of Naturalistic Driving Log Annotations

A Machine Learning Based Approach for Text Classification with a Low Amount of Training Data and Inconsistent Text Fragments

Master's thesis in Systems, Control and Mechatronics + Complex Adaptive Systems

OSCAR BRASK
PERNILLA GELLERMAN

DEPARTMENT OF ELECTRICAL ENGINEERING

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2021
www.chalmers.se

Master's thesis 2021

Multi-Class Text Classification of Naturalistic Driving Log Annotations

A Machine Learning Based Approach for Text Classification with Low Quantity Training Data and Inconsistent Text Fragments

OSCAR BRASK
PERNILLA GELLERMAN

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering
Chalmers University of Technology
Gothenburg, Sweden 2021

Multi-Class Text Classification of Naturalistic Driving Log Annotations

A Machine Learning Based Approach for Text Classification with Low Quantity Training Data and Inconsistent Text Fragment

OSCAR BRASK
PERNILLA GELLERMAN

Supervisor: Mohammad Hossein Moghaddam, Department of Electrical Engineering
Examiner: Prof. Giuseppe Durisi, Department of Electrical Engineering

Master's Thesis 2021
Department of Electrical Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: AD/ADAS visualization of a Volvo Car.

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2021

Multi-Class Text Classification of Naturalistic Driving Log Annotations

A Machine Learning Based Approach for Text Classification with Low Quantity Training Data and Inconsistent Text Fragments

OSCAR BRASK
PERNILLA GELLERMAN

Department of Electrical Engineering
Chalmers University of Technology

Abstract

In this project, a machine learning based algorithm has been developed for multi-class classification of manually written, short driving log text snippets, describing events during Autonomous Drive (AD) and Advanced Driver Assistance Systems (ADAS) testing at Volvo Cars. The algorithm has been divided into three main parts: pre-processing, feature extraction and classification. Many methods for each step have been evaluated and tuned in order to maximize the overall classification performance of all classes. The combination of methods that resulted in highest performance of the algorithm overall was finally selected, which achieved an average F1-score of 0.74. A clear correlation could however be observed between the available amount of training data and the performance of the classification algorithm for each specific category. For future work, a further investigation of the performance obtainable using a higher amount of training data is suggested.

Keywords: Pilot Assist, Text Classification, Text Mining, Machine Learning, Clustering, Natural Language Processing.

Acknowledgements

We would like to thank our adviser Mohammad Hossein Moghaddam at Chalmers University of Technology for his guidance and helpful feedback throughout the project. We would also like to thank our opponents Oskar Nilsson and Marcus Anjemark for supportive feedback. Furthermore, we would like to thank our supervisor at Volvo Cars, Clara Engman, for her great support and continued encouragement in all aspects of the project. Finally, we would like to thank the team at Volvo Cars for the opportunity to carry out this project and the provided resources.

Oscar Brask & Pernilla Gellerman, Gothenburg, May 2021

Contents

Acronyms

ACC Adaptive Cruise Control. 46, 48, 61, 65, 69,
AD Autonomous Drive. v, 1, 3, 51, 65, 67, 69,
ADAS Advanced Driver Assistance Systems. v, 1, 3, 51, 65, 67, 69,
AI Artificial Intelligence. 1, 5, 7, 8, 30,

biLM bi-directional language model. 17, 18,

CBOW continuous bag-of-words. 12, 13,
CMS Collision Mitigation Support. 1, 46, 48, 61, 65, 66, 69,
CNN Convolutional Neural Network. 18, 35–37,
CSA Curve Speed Assist. 46, 48, 61, 65, 66, 69,

DNN Deep Neural Network. 30, 35, 61, 66, 69,

ELMo Embeddings from Language Models. 17–19, 54, 56, 57, 59, 61, 63, 66, 69,

GloVe Global Vectors for Word Representation. 15, 17, 53, 54, 56, 62, 66,

KNN K-Nearest Neighbor. 23–25, 56,

LKA Lane Keeping Aid. 46, 61, 65, 66, 69,
LR Logistic Regression. 57, 61,
LSA Latent Semantic Analysis. 15,
LSTM Long Short Term Memory. 17, 18, 33, 34, 58, 66,

ML Machine Learning. 5, 7, 8,

NB Naive Bayes. 23,
NLP Natural Language Processing. 1, 3, 7–9, 17, 35, 43, 50, 52, 55, 57,

PA Pilot Assist. 1, 46, 61, 65, 69,
PV-DM Paragraph Vectors - Distributed Memory.

RF Random Forest. 56, 61,
RNN Recurrent Neural Network. 32–34, 58, 59, 61, 65, 67,

SVM Support Vector Machine. 25, 27, 59,

TF-IDF Term Frequency-Inverse Document Frequency. 12, 22, 23, 52, 53, 56, 57, 59, 61, 66, 67, 69,
TSI Traffic Sign Information. 1, 46, 51, 61, 65, 69,

1
Introduction

On the journey to a future with driverless and fully autonomous cars, car manufacturers today are competing in the technical development of features within active safety, autonomous drive (AD), and advanced driver assistance systems (ADAS). These technologies have the potential to steer the car autonomously in certain situations, or assist the driver to avoid dangerous situations by actively taking control of the car and for example brake or steer away from an obstacle. However, flaws or problems in these systems could not only decrease their safety improvements, but also cause even worse hazardous situations by forcefully taking actions where not intended. To ensure that the different active safety, AD, and ADAS features work properly, a vast amount of tests of these systems to their limits are needed.

At Volvo Cars, many of these tests are carried out manually, during which text annotations are written for each event during the test. The annotations are saved together with the corresponding sensor data in the form of log files for later analysis. Unfortunately, the text annotations do not follow any standard, and hence often contain misspellings, abbreviations, and different language combinations. These log files are therefore inconsistent in the way they are constructed, which makes the task of automatically sorting them into certain categories very difficult using explicit sorting methods. It is therefore relevant to investigate if this is possible using a machine learning based algorithm instead, in order to categorize a text annotation to a specific active safety, AD, or ADAS feature. Examples of categories of which the text annotations are classified into are Pilot Assist (PA), Traffic Sign Information (TSI) or Collision Mitigation Support (CMS).

1.1 Background

Text classification is a topic within the area of Natural Language Processing (NLP), which is a branch of Artificial Intelligence (AI) that helps computers understand human language. NLP is today used in a huge variety of applications, and a lot of research is still carried out within the area due to its great complexity. Aggarwal and Zhai [1], Kowsari, Jafari Meimandi, Heidarysafa, *et al.* [2] and Mirończuk and Protasiewicz [3] discuss different text classification algorithms in use today and compare their performances. They suggest that the problem of text classification can be broken down into a pipeline consisting of five main parts:

1. **Pre-Processing**. This involves cleaning the text, such as correct misspellings, omit unnecessary characters and words, eliminate slang and abbreviations, as well as handle capitalization.

2. **Feature Extraction**. This involves the complicated procedure of breaking down the unstructured text sequences into a structured feature space, which can be understandable and usable through mathematical modeling. Some of the most common feature extraction techniques in use today are Term Frequency-Inverse Document Frequency [4], Word2Vec [5], and Global Vectors for Word Representation [6], each with different limitations and advantages.

3. **Dimensionality Reduction**. Since text data sets often contain a huge amount of unique words, text classification algorithms can often be computationally expensive and require huge memory space. One common way to reduce this complexity, without compromising the algorithm performance by just using a simpler algorithm, is to reduce the dimensionality of the text. Some of the most common techniques of dimensionality reduction include Principal Component Analysis [7], Linear Discriminant Analysis [8], and Non-Negative Matrix Factorization [9]. Furthermore, there also exist more novel techniques such as Random Projection [10], Autoencoders [11], and T-Distributed Stochastic Neighbor Embedding [12].

4. **Classification**. This is the main part of a text classification algorithm, and also the most important step. A huge variety of classification algorithms exist, with different applicable use cases. Some examples are Rocchio Classification [13], ensemble-based learning techniques such as Boosting and Bagging [14], simple Logistic Regression [15], the Naïve Bayes Classifier [16], K-Nearest Neighbor [17], Support Vector Machines [18], tree-based classifiers such as Decision Trees [19] and Random Forest [20], and most recently, Deep Learning [21].

5. **Evaluation**. This is the final part of the pipeline, which is important in order to understand how the model performs and to enable comparisons between different methods in a fair way. Some examples of evaluation metrics are regular Accuracy [22], F1-score [22], Matthews Correlation Coefficient [23], and Receiver Operating Characteristics [24].

However, not all the different methods and techniques mentioned in this pipeline will be applicable for this project. First of all, since the team at Volvo Cars wants to determine and label the categories themselves, this project will focus on supervised- or semi-supervised learning, where the training algorithm will be given labeled pairs of a text phrase and the category it belongs to. Therefore, all cases of fully unsupervised clustering will not be applicable for this project. Furthermore, many commonly used text classification algorithms require the text data to be in the form of documents, paragraphs or at least complete sentences in order to perform well. In this project, the text annotations consist of short comments built up by only a few words, which causes many algorithms to not be easily applicable.

This entails extensive limitations in the methods available. However, some research in the area of short text classification has been performed. For example, Zeng, Li, Song, *et al.* [25] evaluate the usage of Topic Memory Networks, and Wang, Wang, Zhang, *et al.* [26] use the combination of Deep Convolutional Neural Networks and a taxonomy knowledge base for the text. Their results are promising, but their applicability in this project is limited due to the amount of training data required by these techniques. The reason is that, in this project, the available amount of labeled annotations to be used as training data is very limited. Hence, many deep learning related methods are most likely impractical to implement, since they usually require vast amount of examples in order to achieve satisfactory performance.

In other machine learning cases where training data is limited, a common approach is to use Transfer Learning. This involves using a pre-trained model, which has been trained on a very large data set beforehand, but for a different problem in the same area. By tuning this pre-trained model, for example by adding layers or exclusively training the top layers given that the model is based on Deep Learning, it can be customized to fit the desired problem. Howard and Ruder [27] have been evaluating the transfer learning approach for the problem of text classification, by developing a Universal Language Model applicable to any NLP and fine tuning the model to fit their specific case. Their results are promising, and with only 100 training examples they succeeded to outperform traditional state-of-the-art classification models. However, these performance metrics were made on rather large text documents or paragraphs, and the feasibility for our case of short sub-sentences or comments is unclear.

It can therefore be concluded that previous efforts have been made in the problem area of developing a text classification algorithm both for the case where the text is in the form of very short, unorganized comments, as well as for the case where the specific training data is very limited. However, very little research for the problem of short, grammatically incorrect and misspelled text snippets in combination with a very limited amount of training data is available. Hence, this problem is scientifically relevant and deserves further investigations.

1.2 Purpose

The purpose of this project is to develop and evaluate a text classification algorithm, trained on a proportionally small training data set compared to the total driving log data collection. This evaluation is executed in order to enable development of an efficient tool that can identify key features in short text comments, and help Volvo Cars to classify their log files into different Active Safety- AD- or ADAS function categories. The execution revolves around developing a Python script, which given a training data set of labeled annotations can learn specific features in the text in order to be able to categorize new, unlabeled text annotations.

1.3 Objective

The purpose above can be broken down into a number of main research questions, which are the following:

- How can a text classification algorithm be constructed such that it can successfully be applied to classification problems in which the training data is limited and the text annotations to classify are short and unstructured?
- How can the different text classification algorithms be compared regarding their performance?
- What are the key model features of such algorithms that yield different performance results?

1.4 Scope

The project covers both a literature study and a hands-on implementation of different classification algorithms in order to ultimately determine which approach can predict the correct classes of the annotation comments. Due to the structure of these manual text annotations, the text classification algorithm needs to be tuned in order to handle short wordings rather than complete sentences or paragraphs. Moreover, the restricted amount of available training data needs to be taken into consideration when selecting machine learning approaches.

2
Theory

This chapter is dedicated to clarify the underlying concepts of machine learning in general and text classification in particular. The section covering machine learning briefly explains the fundamentals of the field as a whole, but focuses more specifically on three different categories: supervised-, unsupervised- and semi-supervised learning. Moreover, the text classification is broken down into four segments, each covering more in-depth information extracted from previous research. Within each classification subsection, the already existing methods are scrutinized in order to gain knowledge of the algorithms that are suitable for solving this particular classification problem of the driving log annotations.

2.1 Machine Learning

The field of AI contains subareas representing different approaches for development of systems that can interpret and learn from data. This knowledge can thereafter be used by the system to solve tasks and reach specific goals [28]. One of the AI subfields is machine learning (ML), which is an essential part of AI with its methods that support the system to learn, without explicit programming, about patterns and more complex information within the data. Mitchell [29] gives a definition of what *learning* implies: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E". Goodfellow, Bengio, and Courville [30] elaborate on the interpretation of task T, performance P, and experience E. They suggest that ML enables the computer to solve tasks that are unsolvable using explicitly fixed programs, implying that the task is simply not the learning process but rather the given assignment, which the system is set to work out. This could for example be tasks within classification, machine translation, or system anomaly detection. To determine how well the computer manages to solve the given problem, there is a need for implementing a quantitative performance measurement. Three examples of measurements are accuracy, error rate, and log-probability. The accuracy is simply the proportion of correct outputs, and on the contrary, the error rate is the proportion of incorrect outputs. When the answer cannot be determined in a binary fashion, the average log-probability that the system gives to some examples can typically be used. The evaluation is generally done on data that is previously unseen by the computer, which gives an indication of how successful the algorithm will be when implemented on subsequent similar data. To gain great performance, the system has to gather experience through encounters

with task-specific data. Depending on the type of experience that the machine learning methods are limited to during the learning process, they can be sorted into three types of learning categories: supervised learning, unsupervised learning and reinforcement learning. However, reinforcement learning is mostly applied to cases including environments and unfixed data sets. Hence, the following subsections exclusively clarify the basis for supervised-, unsupervised-, and the combination of the two; semi-supervised learning.

2.1.1 Supervised Learning

In the most classic cases of machine learning, an algorithm is fed with N number of training examples with corresponding labels [31]. These can be mathematically expressed as $\{(\boldsymbol{x_1}, y_1), \ldots, (\boldsymbol{x_N}, y_N)\}$ such that $\boldsymbol{x_i}$ is the feature vector of the ith training example and y_i is its corresponding label or class. By observing the training data pairs, the model uses inductive reasoning to generalize a mapping function $f : \boldsymbol{X} \rightarrow \boldsymbol{Y}$ between the training examples $\boldsymbol{X_{train}}$ and their labels $\boldsymbol{Y_{train}}$. The model then uses this learned mapping rule to be able to predict the labels $\boldsymbol{Y_{test}}$ on unseen test data $\boldsymbol{X_{test}}$. This is called supervised learning, since the output of the training data is known by the algorithm.

2.1.2 Unsupervised Learning

Another category of machine learning algorithms is unsupervised learning, which is used to learn patterns from N number of untagged training data examples, $\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_N} \in \boldsymbol{X}$, which has not been labeled or classified [30]. The algorithm does not have any prior knowledge about the desired labels $\boldsymbol{Y}$, and instead uses properties of the data to recognize certain features. The outcome could either be to learn a probability distribution, or as in the case of classification problems, to cluster the data into different distinguishable clusters, as illustrated in Figure 2.1.

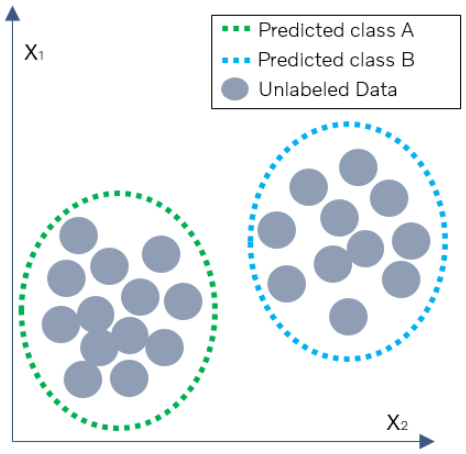
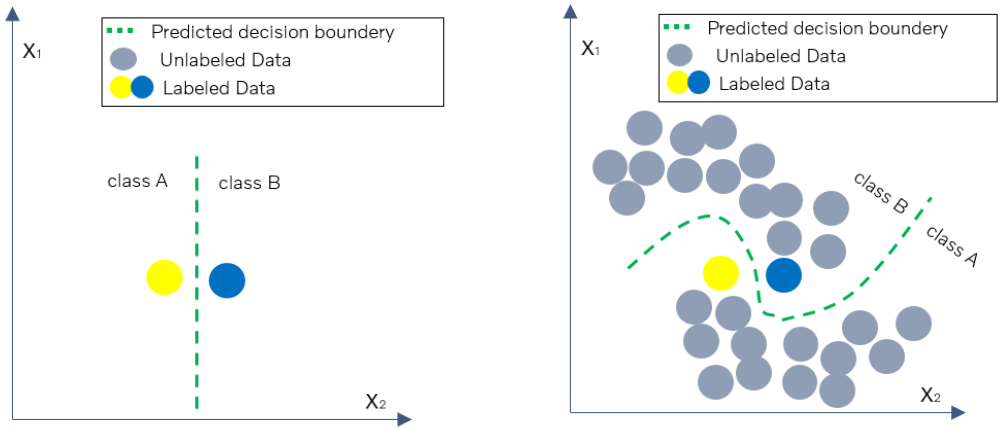


Figure 2.1: A clustering example of unsupervised learning, where unlabeled data with features x_1 and x_2 are classified into two classes A and B.

2.1.3 Semi-supervised Learning

When both labeled and unlabeled data are used in combination, the learning strategy is called semi-supervised learning [32]. The idea is to use a small amount, l, of training examples $\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_l} \in \boldsymbol{X}$ with corresponding labels $\boldsymbol{y_1}, \boldsymbol{y_2}, \ldots, \boldsymbol{y_l} \in \boldsymbol{Y}$, in combination with a larger amount, u, of unlabeled training examples $\boldsymbol{x_{l+1}}, \boldsymbol{x_{l+2}}, \ldots, \boldsymbol{x_{l+u}} \in \boldsymbol{X}$. This eventually increases the algorithm's performance compared to the case where either labeled or unlabeled data is used exclusively. This is illustrated in Figure 2.2. Furthermore, manually labeled data, as in the supervised learning case, could potentially be very expensive and difficult to gather. A semi-supervised approach could therefore be a way to increase the accuracy of algorithms without the high effort of manually labeling more data. In the classification case, the basic concept is an unsupervised learning strategy that is enhanced by a small proportion labeled data. However, using semi-supervised learning often requires certain data properties to be fulfilled, such that the data tend to form discrete clusters, and that the data points in the same cluster are more likely to share a label.



(a) A clustering example of supervised learning, where two labeled data examples are classified into two classes A and B.

(b) A clustering example of semi-supervised learning, where labeled and unlabeled data in combination are classified into two classes A and B.

Figure 2.2: Two clustering examples of data with features x_1 and x_2, which demonstrate how the decision boundary could improve by adding unlabeled training examples to a supervised classification case.

2.2 Text classification

Two of the subfields of AI are machine learning (ML) and natural language processing (NLP), which relationships to AI and each other can be seen in Figure 2.3. As illustrated, the two subfields overlap. This combination of ML and NLP to learn linguistic structures has gained increasing attention since the early 90's [33]. The

text classification problems belong to a branch of NLP, and can be either machine learning based or not. However, most of the classification methods covered in this section are in the machine learning space due to the nature of the project.

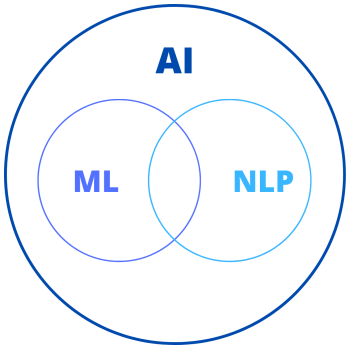
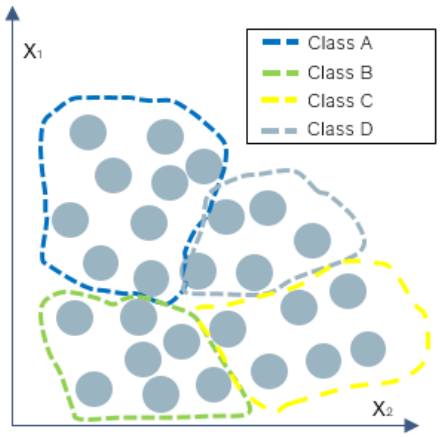
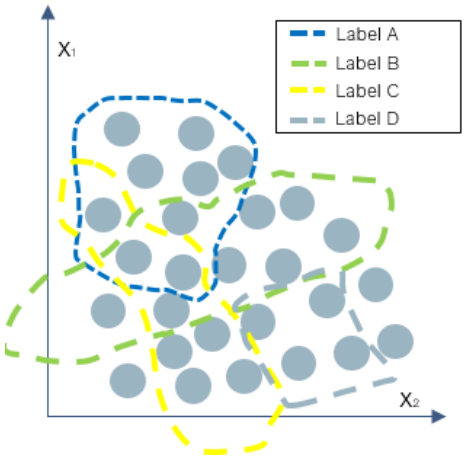


Figure 2.3: The relationship between the fields of AI, ML, and NLP.

Another way of separating text classification tasks is the labeling strategy. If each class of the data is easily distinguishable and mutually exclusive, the task is called multi-class text classification [34]. Each data example must belong to one class, and one class only. This is different from the other type of classification, called multi-label classification, where the algorithm considers each label as a binary classification problem, also known as the one-vs-all type classification strategy. Therefore, each data example can belong to several labels at the same time, or even none. This is illustrated in Figure 2.4.



(a) A multi-class classification example where the data are classified into either of the four classes A, B, C, and D.



(b) A multi-label classification example where the data are given four labels A, B, C, and D.

Figure 2.4: Two classification examples of the same data with features x_1 and x_2, which demonstrate the difference between a multi-class and a multi-label approach.

Kowsari, Jafari Meimandi, Heidarysafa, *et al.* [2] have, as mentioned above, proposed a pipeline that many text classification algorithms tend to follow. This pipeline has been broken down into different parts, and below each part is presented in-depth.

2.2.1 Pre-processing

The first step in many NLP algorithms is pre-processing of the text, which facilitates the later learning process of the classification algorithm [35]. The pre-processing step can be broken down into several sub-steps, each described in more detail below.

2.2.1.1 Noise Removal

Some of the content in text strings, such as punctuation marks, numbers, and characters outside of the alphabet, increases the complexity of the classification algorithm without affecting the performance [36]. Therefore, these are usually treated as noise and are simply removed from the text data. Furthermore, contractions such as "isn't", "can't" etc, are often replaced with their longer forms.

2.2.1.2 Slang and Abbreviations

Slang and abbreviation are other forms of text anomalies which could confuse the classification algorithm, and are therefore often converted to regular language. An abbreviation is a shortened form of a word or phrase, which contains mostly upper case first letters from the included words. Therefore, abbreviations are often easy to identify, and can be handled by extending them to regular words with the usage of a simple dictionary. Slang expressions, on the other hand, are much more complex to identify. Expressions like "screw up", "something is lit", or "oh my god" etc, are often used with a totally different sense than their literal meaning. However, Dhuliawala, Kanojia, and Bhattacharyya [37] have developed a resource *SlangNet*, which is a lexical database containing 3000 English slang words and their semantics. Many similar efforts exist, and the usage of such slang lexicons in NLP algorithms tend to noticeably improve the algorithm performance [38].

2.2.1.3 Capitalization

Since text often consists of both uppercase and lowercase letters, the complexity and number of variations of each word is often unnecessary high. Therefore, a simple step to decrease the number of variations of words with the same meaning is to convert all letters to lowercase. However, it is important to do this conversion after the abbreviation step above is completed. This since word abbreviations like "US" (United States of America) have a totally different meaning compared to the lowercase "us", which then could be missed by the abbreviation detector [38].

2.2.1.4 Stop-words removal

Usually, the text data contain many words without any important information contribution to the classification. These are called stop-words, and are often completely removed from the text, since this tends to ease the later classification process [39]. Examples of stop-words are "a", "about", "above", "across", "after", "afterwards", and "again".

2.2.1.5 Spell Correction

A crucial step in the text pre-processing is to implement spell correction, so that potential typographical errors are corrected before the next parts of the algorithm are executed. One common approach is to use the so called edit distance method [40]. It is a method which finds the closest correct word to a misspelled one by calculating the minimum edit efforts, treated as operation costs, that have to be made in order to convert the misspelled word into a correct one found in a dictionary. There are however many types of edit distance metrics, of which the most commonly known is the Damerau–Levenshtein distance [41]. To express the Damerau–Levenshtein distance between two strings a and b, a function $d_{a,b}(i,j)$ is defined, whose value is the distance between an i-symbol prefix of string a and a j-symbol prefix of string b. Furthermore, four possible edit actions are defined:

- *Deletion:* When a letter is deleted, for example classifi**cc**ation → classifi**c**ation.
- *Insertion:* When a letter is being inserted, for example class**ii**cation → class**ifi**cation.
- *Substitution:* When one letter is being replaced by another, for example classifi**ka**tion → classifi**ca**tion.
- *Transposition:* When two letters next to each other are changing place, for example classifi**ac**tion → classifi**ca**tion.

Each of these edit actions is then given an expression of what Damerau–Levenshtein distance they correspond to, according to the following:

- *Deletion:* $d_{a,b}(i-1,j)+1$
- *Insertion:* $d_{a,b}(i,j-1)+1$
- *Substitution:* $d_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)}$
- *Transposition:* $d_{a,b}(i-2,j-2)+1$

Putting these together, the full equation for the Damerau-Levenshtein distance between two strings a, b is given by $d_{a,b}(|a|,|b|)$, where $d_{a,b}(i,j)$ is calculated recursively as specified in (2.1):

$$d_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j)=0, \\ \min\begin{cases} d_{a,b}(i-1,j)+1 \\ d_{a,b}(i,j-1)+1 \\ d_{a,b}(i-1,j-1)+1_{(a_j \neq b_j)} \\ d_{a,b}(i-2,j-2)+1 \end{cases} & \text{if } i,j>1 \ \&\ a_i = b_j - 1 \ \&\ a_i - 1 = b_j, \\ \min\begin{cases} d_{a,b}(i-1,j)+1 \\ d_{a,b}(i,j-1)+1 \\ d_{a,b}(i-1,j-1)+1_{(a_j \neq b_j)} \end{cases} & \text{otherwise.} \end{cases} \tag{2.1}$$

2.2.1.6 Stemming & Lemmatization

In natural language, many variations of words with the same semantic meaning often exist. One method for consolidating different forms of a word into the same feature space is to use stemming or lemmatization [42]. For example, word variations like "great", "greatly", "greatest", and "greater", do all have the same meaning and

would by the usage of stemming or lemmatization be converted to their root word. This decreases entropy and increases the relevance of the concept “great”, which facilitates the computation process [43]. Stemming is the easiest implementation of root word conversion, and is performed by simply removing the suffixes “ly”, “est”, and “er”, so that the root word “great” is left. However, this is done completely without knowledge of the context. Therefore, stemming cannot differentiate between words like the noun “meeting” and the verb “meet”. Lemmatization, on the other hand, do take this context into consideration by using a context aware dictionary of word lemmas. This is a more challenging task and has therefore higher accuracy, at the cost of higher computational complexity. Using stemming or lemmatization has in several studies [35], [44] increased the performance of the text classification algorithm significantly, if compared to the case of unprocessed text.

2.2.2 Feature extraction

One of the more complicated steps in a text classification pipeline is the extraction of mathematically understandable features. Some of the feature extraction techniques also entail dimensionality reduction, although the number of dimensions are commonly reduced in a consecutive step to the feature extraction [2]. The main purpose of the feature extraction is to determine and select the words that are essential for the interpretation of the text, which entails a mathematically structured feature space.

Most of the following methods within feature extraction belongs to either the word-embedding- or term-weighting category. Word-embedding methods focus on feature learning and use a N dimensional vector of real numbers to map each word or phrase of the vocabulary. Words that are close semantically usually have similar matrix representations [45]. Term-weighting techniques, on the other hand, use a vector containing the assigned weights of each word in the corresponding document, in order to distinguish the terms that are of importance [46]. These term-weighting systems are meant to increase the retrieval effectiveness and are evaluated on how well they extract the relevant terms and reject the extraneous terms, which correspond to measures called *recall* and *precision* respectively [4]. The following method descriptions clarify how each feature extraction technique works and specifically which category they belong to.

2.2.2.1 Term Frequency-Inverse Document Frequency (TF-IDF)

The conflicting part of a term-weighting system is to balance the performance of both recall and precision [4]. The recall measure is based on the amount of relevant retrieved items out of the total number of relevant items in the whole text data set, which calls for a usage of terms that are broad and frequently used in a large proportion of the text documents. On the contrary, a high precision is based on the amount of relevant retrieved items compared to the total number of retrieved items. Therefore, more distinct and narrow words increase the chances of better precision effectiveness. Due to these different term requirements, it is beneficial to use term

weighting factors that take advantage of both the recall- and precision-improving elements.

Term frequency (TF) is a factor that can improve the recall part of the retrievals through distribution of a value to every individual term, which corresponds to their occurrence frequency within the text corpus. Words that appear often in the documents receive higher weights and are therefore seen as more important than less recurring ones. However, only considering the repetitiveness of words can compromise the overall performance of the term-weighting system, since this might yield a document selection including loads of non-relevant entities. Hence, another factor should be implemented. *Inverse document frequency* (IDF) assigns weights inversely to the proportion of documents that the term exists in. Jones [47] suggests that the IDF factor can be calculated through

$$\text{factor}_{\text{idf}} = \log \frac{N}{n}, \tag{2.2}$$

where n is the number of documents that the term is related to and N is the size of the whole document collection. Combining the two frequency factors through multiplication, $\text{factor}_{\text{tf}} \cdot \text{factor}_{\text{idf}}$, yields the term-weighting system Term Frequency-Inverse Document Frequency (TF-IDF) [4]. This factor merge is done as an attempt to distinguish the words with high term frequency but an overall low document collection frequency. Salton and Buckley [4] also introduce a normalization factor in order to manage documents of largely varying lengths. Shorter documents are usually represented by shorter term vectors compared to vectors embodying longer texts, which implies that the search query has a higher chance of matching documents of greater size. Ultimately, this can work in favor of the larger documents during the retrieval phase. Therefore, normalizing the weights within each term vector as shown in (2.3) or (2.4) helps to reduce the level of bias during retrieval.

$$w_{normalized} = \frac{w}{\sum_{\text{i}} w_i} \tag{2.3}$$

$$w_{normalized} \quad \frac{w}{\sqrt{\sum_{\text{i}} (w_i)^2}} \tag{2.4}$$

2.2.2.2 Doc2Vec

In 2013 Mikolov, Chen, Corrado, *et al.* [48], [49] presented a new word-embedding method that was turned into a software called Word2Vec. It was created as an attempt to learn high-quality word vectors from enormous data sets, containing billions of words, with a vocabulary consisting of millions of words. Word2Vec is based on three concatenated structures: neural networks with two hidden layers, the continuous bag-of-words (CBOW), and the continuous Skip-gram model. The neural networks solely work as a base for the construction of both the CBOW and the continuous Skip-gram model. The two different structures are less complex versions of earlier proposed neural network language models, which consist of an input layer, a projection layer, a hidden layer, and an output layer. The Word2Vec

model operates on neural networks with an input layer, a projection layer, and an output layer, thus removing the hidden layer due to the increased computational complexity that it entails. Each corresponding neural network structure is shown in Figure 2.5.

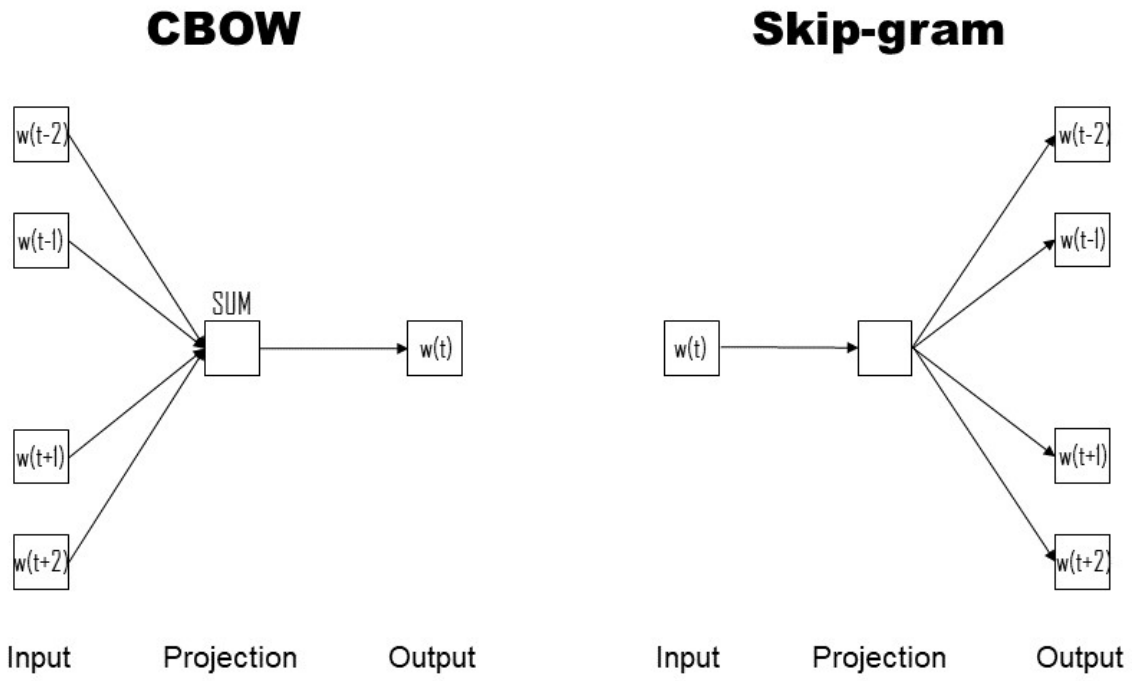


Figure 2.5: A schematic diagram of the neural networks used by the CBOW and the Skip-gram model to calculate the current word based on the context and surrounding words given the current word, respectively.

The Skip-gram model calculates predictions of the surrounding words based on the current one. Hence, its objective is to learn the word vector representations that maximize the average log probability in (2.5), where T is the total number of words in a given training sequence, c is the size of the training context, i.e., the number of words before and after the current word, and w is a word.

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c, j\neq 0}\log p(w_{t+j}|w_t) \tag{2.5}$$

The network for CBOW is designed to predict the opposite, namely the current word based on an input consisting of a number of preceding and following words. Thus, deducing the current word based on the surrounding context. All the input words use the same weight matrix and are projected into the same position, where their vectors are averaged, which is the reason behind the name CBOW. The bag-of-words name symbolizes the fact that the order of words in the input does not affect the projection. The objective is to maximize the average log probability seen in (2.6), where p is the probability and T is the number of training words in the sequence [50]. Typically, the following step includes a multi-class classifier, such as

the softmax that relates to (2.7), to predict the word.

$$\frac{1}{T}\sum_{t=k}^{T-k}\log p(w_t|w_{t-k},\ldots,w_{t+k}) \tag{2.6}$$

$$p(w_t|w_{t-k},\ldots,w_t+k) = \frac{e^{y_{w_t}}}{\sum_i e^{y_i}} \tag{2.7}$$

The factor y_i is an unnormalized log probability for each output word, i, which is calculated as

$$y = b + Uh(w_{t-k},\ldots,w_{t+k};W), \tag{2.8}$$

where b and U are the softmax parameters; h is a concatenation or average of the word vectors extracted from the word vector matrix W, which is built up by unique vectors mapped to a word each.

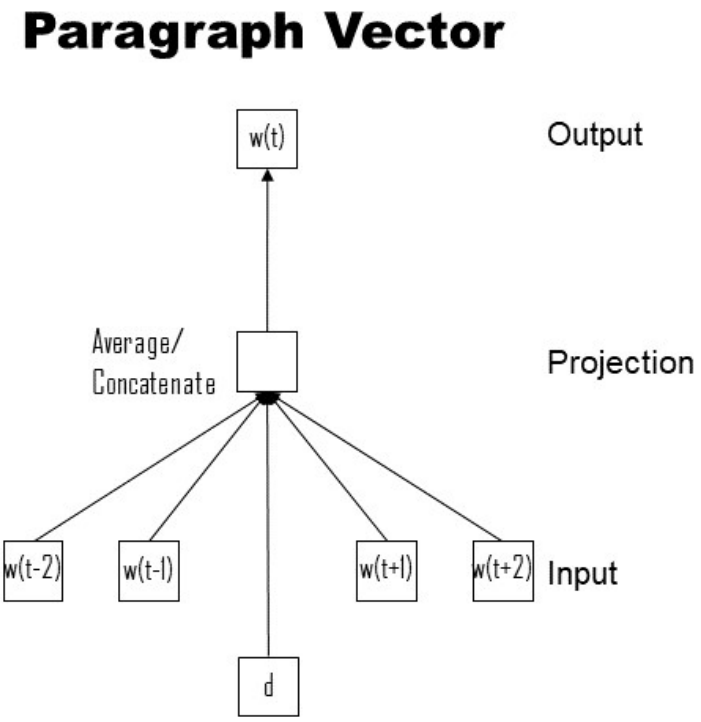


Figure 2.6: A schematic diagram of the neural networks used for calculating the paragraph feature representations in the Doc2Vec model.

As an extension of the Word2Vec model, the *Paragraph Vector*, also called Doc2Vec, was introduced in 2014 to enable representations of variable-length texts through fixed-length feature vectors [50]. The texts can be short sentences, paragraphs or large documents. Training of the Paragraph Vector is similar to that of Word2Vec, where the word vectors are randomly initialized and learned in an unsupervised fashion. However, in addition to every word being represented by a unique vector in the matrix W, each paragraph as a whole is also mapped to a unique vector in matrix D. The structure of the Doc2Vec network can be seen in Figure 2.6. The new

structure only yields a single computational difference, namely the calculations of h, which is constructed from both W and D in the Doc2Vec framework. The vector from matrix D can be seen as a memory vector that contains information about the whole paragraph, such as the topic. Hence, the model is often referred to as the Distributed Memory Model of Paragraph Vectors (PV-DM). Each vector from matrix D is paragraph specific and is used within a paragraph to generate contexts, whereas the vectors in W are used globally and shared across all paragraphs.

2.2.2.3 Global Vectors for Word Representations (GloVe)

Pennington, Socher, and Manning [6] introduces a similar method to Word2Vec called Global Vectors for Word Representation (GloVe), which also utilizes high dimensional vectors to represent words. GloVe is based on a global log-bilinear regression model, that in turn is a combination of the two dominant method categories: local context window methods and global matrix factorization. The previously described Skip-gram model is a type of local context window method. An example of global matrix factorization is the latent semantic analysis (LSA), which looks at the implicit high-order structures that relate certain words to specific documents [51]. As the method category implies, the information is stored in a large matrix, more specifically a term-by-document matrix. The matrix can be decomposed through singular-value decomposition (SVD) into approximately 100 orthogonal factors, which then can be combined linearly to form the original matrix. A document can consequently be embodied by combining roughly 100 item vectors of factor weights. Hence, words in queries can form weighted combinations that embody pseudo-document vectors.

The GloVe method [6] aims to combine the two method families in order to avoid the shortcomings of each algorithm type respectively. Pennington, Socher, and Manning suggest that LSA include a sub-optimal vector space due to its inability to recognize word similarities on a satisfactory level. However, it does perform well on leveraging statistical information about documents. On the other hand, the local context window methods are better at analogy tasks, while they miss to take advantage of the often repetitive wordings in text data. Thus, methods that focus on small, local text segments generally do not utilize the corpus statistics in an efficient way. The GloVe model avoids the statistics retrieval drawbacks by using a word-word co-occurence matrix and only train on the non-zero elements. Thus, keeping the analysis on a global level rather than on individual small context windows, and disregarding the empty slots in the large sparse matrix for higher efficiency. More specifically, it is trained using a peculiar weighted least squares model, which ultimately outputs a vector space containing a relevant sub-structure. The sub-structure of the vector space helps GloVe hold a state-of-the-art performance with its 75% accuracy on the provided word analogy data set. Moreover, it outperforms several other models on different word analogy tests and on a common named entity recognition benchmark called CoNLL-2003, which is further described by Sang and De Meulder [52].

In order to obtain the equations needed to train the GloVe vectors, Pennington, Socher, and Manning [6] describe the details in the method as follows. A co-

occurrence count matrix X is defined, whose entries X_{ij} tabulate the number of times a word j occurs together in the context of another word i in the corpus. The number of times any word appears in the context of word i can then be denoted as $\sum_k X_{ik}$, and the probability that word j appear in the context of word i as $P_{ij} = P(j|i) = X_{i,j}/X_i$. The probability ratio P_{ik}/P_{jk} will then be a representation of similarity between words. Some examples are given in Table 2.1 for the words *i="ice", j="steam"* and different words for k.

Table 2.1: An example of co-occurence probabilities for target words *ice* and *steam* with selected context words. Source: [6].

Probability and Ratio	k = *solid*	k = *gas*	k = *water*	k = *fashion*
$P(k\|ice)$	1.9×10^{-4}	6.6×10^{-5}	3.0×10^{-3}	1.7×10^{-5}
$P(k\|steam)$	2.2×10^{-5}	7.8×10^{-4}	2.2×10^{-3}	1.8×10^{-5}
$P(k\|ice)/P(k\|steam)$	8.9	8.5×10^{-2}	1.36	0.96

It is possible to observe from Table 2.1 that when k represents a word unrelated to both *ice* and *steam*, for example *k="water"* or *k="fashion"*, the ratio gets close to 1. When k instead represents a word unrelated to *steam* but related to *ice*, such as *k="solid"*, the ratio is $>> 1$. And vice versa, in the case when when k represents a word very similar to *steam* but irrelevant to *ice*, such as *k="gas"*, the ratio is $<< 1$.

It therefore possible to conclude that the ratio of co-occurrence probabilities P_{ik}/P_{jk} is a powerful way to distinguish relevant words from irrelevant words in the global corpus, compared to using the probabilities themselves. A function F which describes this ratio is therefore constructed as follows:

$$F(w_i, w_j, \hat{w_k}) = \frac{P_{ik}}{P_{jk}} \tag{2.9}$$

in which w are the two word vectors and $\hat{w}$ is a context word vector. Since word vectors are linear systems, and since only the difference between word i and j are of interest, (2.10) can be written as:

$$F(w_i - w_j, \hat{w_k}) = \frac{P_{ik}}{P_{jk}} \tag{2.10}$$

However, (2.10) has vectors on the left-hand side and a scalar on the right-hand side, which needs to be handled. This is done using the dot product:

$$F((w_i - w_j)^T \hat{w_k}) = \frac{P_{ik}}{P_{jk}} \tag{2.11}$$

The next step is then to ensure the concept of a symmetrical relationship between the similarity of two words. This enforces F to have the property:

$$F((w_i - w_j)^T \hat{w_k}) = \frac{F(w_i^T \hat{w_k})}{F(w_j^T \hat{w_k})} \tag{2.12}$$

which together with (2.11) is solved by:

$$F(w_i^T \hat{w}_k) = P_{ik} = \frac{X_{ik}}{X_i} \tag{2.13}$$

By studying (2.12), it is possible to see that $F = exp$ is a valid solution. Hence, using (2.13):

$$w_i^T \hat{w}_k = \log(P_{ik}) = \log(X_{ik}) - \log(X_i). \tag{2.14}$$

Equation (2.14) however, does not preserve the symmetrical relationship. This can be solved by absorbing the term $\log(X_i)$ into an introduced bias b_i for w_i, which together with an additional bias $\hat{b}_k$ for $\hat{w}_k$ forms the final expression:

$$w_i^T \hat{w}_k + b_i + \hat{b}_k = \log(X_{ik}) \tag{2.15}$$

The final step in the GloVe pipeline is to construct a cost function J, to be able to train the word vector representations of weights and biases such that equation 2.15 is fulfilled. Introducing a weighting function $f(X_{ij})$, this is done as follows:

$$\begin{aligned} J &= \sum_{i,j=1}^{V} f(X_{ij})(w_i^T \hat{w}_j + b_i + \hat{b}_j - \log(X_{ij}))^2 \\ f(x) &= \begin{cases} (x/x_{\max})^\alpha & \text{if } x < x_{\max} \\ 1 & \text{otherwise.} \end{cases} \end{aligned} \tag{2.16}$$

where $x_{\max}$ and α are constant scalars.

2.2.2.4 Contextualized Word Representations (ELMo)

Peters, Neumann, Iyyer, *et al.* [53] introduce another word embedding technique, namely a deep contextualized word representation. It aims to model both the more complex characteristics of words, such as semantics and syntax, as well as how polysemous words, i.e., words with two or more meanings, are used depending on the context. Among other models, the unsupervised *context2vec* [54] acts as a based for the contextualized word representations technique. It is built on a bidirectional Long Short Term Memory (LSTM), which helps to encode the context around a specific word. The vectors used for the deep contextualized word representations are derived from a bidirectional LSTM, that is trained on a large text corpus with a coupled language model (LM) objective [53]. Differing from earlier work, each token has a representation based on the whole input sentence. Hence, they are Embeddings from Language Models, or ELMo representations. These representations of the embeddings are deep compared to those in previous learning concepts for contextualized word vectors, such as [55] and [56], since they are a function of all internal layers of the two bi-directional language model (biLM) layers, instead of only the top LSTM layer. This structure enables semi-supervised learning for pretraining of the biLM at a generous scale, and the resulting representations can be applied to already existing neural NLP architectures to improve their performance on a wide range of/ NLP challenges.

The biLM consists of two parts: a forward and a backward LM. A forward LM computes the probability of the token sequence $(t_1, t_2, ..., t_N)$ through modeling the probability of token t_k, given the past tokens $(t_1, ..., t_{k-1})$. On the contrary, a backward LM uses the future context to predict the previous tokens.

$$p_{\text{fw}}(t_1, t_2, ..., t_N) = \prod_{k=1}^{N} p(t_k | t_1, t_2, ..., t_{k-1}) \tag{2.17}$$

$$p_{\text{bw}}(t_1, t_2, ..., t_N) = \prod_{k=1}^{N} p(t_k | t_{k+1}, t_{k+2}, ..., t_N) \tag{2.18}$$

By combining (2.17) and (2.18) into (2.19), the log probability for both directions can be jointly maximized. The parameters for the token representation (Θ_x) and the Softmax layer (Θ_s) are shared between the directions, while the LSTM parameters are kept separately. These equations stand as a base for further calculations of the ELMo vectors, that are task specific weightings of the biLM layers, which can be studied in depth in the work of Peters, Neumann, Iyyer, *et al.* [53].

$$\begin{aligned} \sum_{k=1}^{N} (& \log p(t_k | t_1, ..., t_{k-1}; \Theta_x, \overrightarrow{\Theta}_{LSTM}, \Theta_s) + \\ & \log p(t_k | t_{k+1}, ..., t_N; \Theta_x, \overleftarrow{\Theta}_{LSTM}, \Theta_s)) \end{aligned} \tag{2.19}$$

A schematic view of the ELMo algorithm is shown in Figure 2.7. It works as follows:

- Given a text annotation, consisting of a number of words, each word is fed through a character-level Convolutional Neural Network (CNN) which outputs raw word vectors. The CNN architecture is explained in more detail in Section 2.2.3.8.3.
- These are then inputs to the first ELMo layer, consisting of two Long Short-Term Memory (LSTM) passes: one with backward connected LSTM cells to represent the context of the words after that word, and one with forward connected LSTM cells to represent the context of the words before that word. The LSTM architecture is explained in more detail in Section 2.2.3.8.2. This pair of information from the forward pass and backward pass then form intermediate word vectors.
- These are fed into the next layer of biLM, after which a second stage of intermediate word vectors are formed.
- The final representation of ELMo is the weighted sum of the two intermediate word vector stages and the raw word vectors. This gives one numerical 1-dimensional vector for each word, and inputting a sentence with several words will result in a 2-dimensional output.

2.2.3 Classification

When the text has been pre-processed and extracted to numerical feature vectors, the main part of the pipeline is then to perform the classification process itself. If the problem is multi-class classification, the algorithm will assign each document to be classified into one category. If the task instead is a multi-label approach,

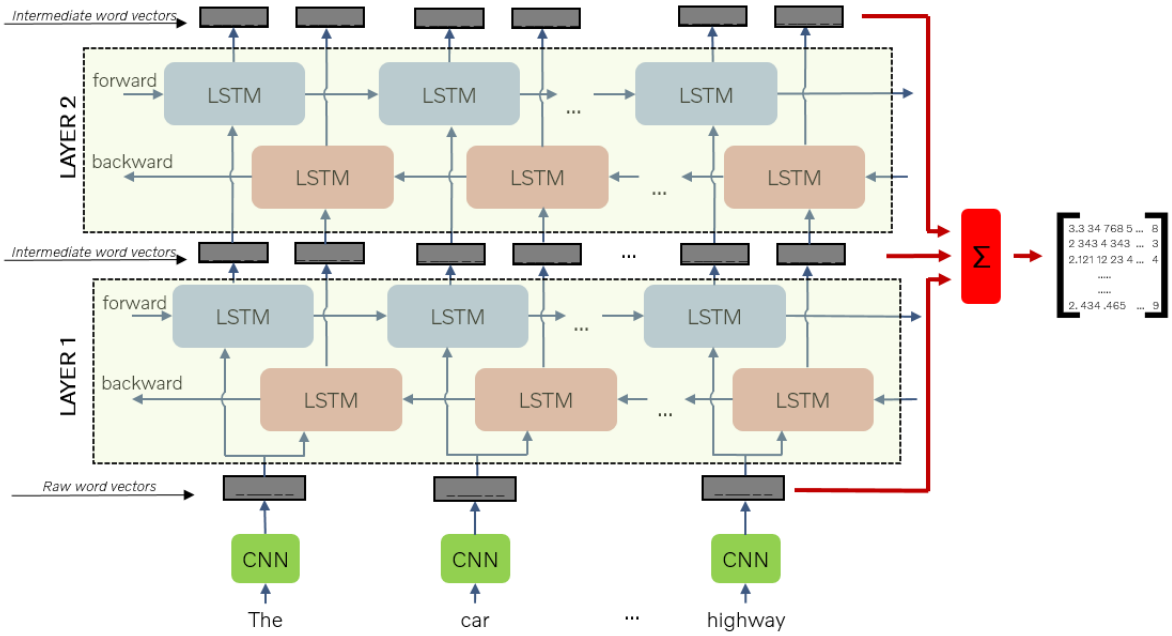


Figure 2.7: A schematic view of the ELMo algorithm for feature extraction.

each document can be assigned to zero, one or several classes. In a supervised- or semi-supervised case, these classes are known beforehand, while the algorithm in an unsupervised case instead decides the classes itself. [57].

Many classification algorithms exist. In this section, the most common machine-learning based approaches will be described, each with different pros, cons, and limitations.

2.2.3.1 Bagging

The general idea of combining underlying algorithms in a multi-model architecture is called ensemble classifier, and has been used successfully for supervised text classification problems. One ensemble classification method is bagging [58], a term which comes from the words **B**ootstrap **Agg**regator. Bootstraping is a general procedure within supervised machine learning, which can be used to improve the performance of an algorithm with high variance, such as decision trees. This is done by creating small multiple samples of data from an entire dataset, drawn with replacement. This bootstrap part of the bagging algorithm works by ensembeling, and will fit the base learning algorithms on each randomly created subset from the original data. The aggregator part of the algorithm then combines the predictions of all the learners by voting or averaging their outputs to get the final results, which eventually reduces variance and give smoother predictions compared to the case where a single classifier is used. Different from Boosting [59], which is another ensemble method that improves the classification performance by handling the underlying algorithms in sequence, the bagging procedure instead works in parallel [60]. This is shown in Figure 2.8. The pseudo-code for the bagging method is presented in algorithm 1.

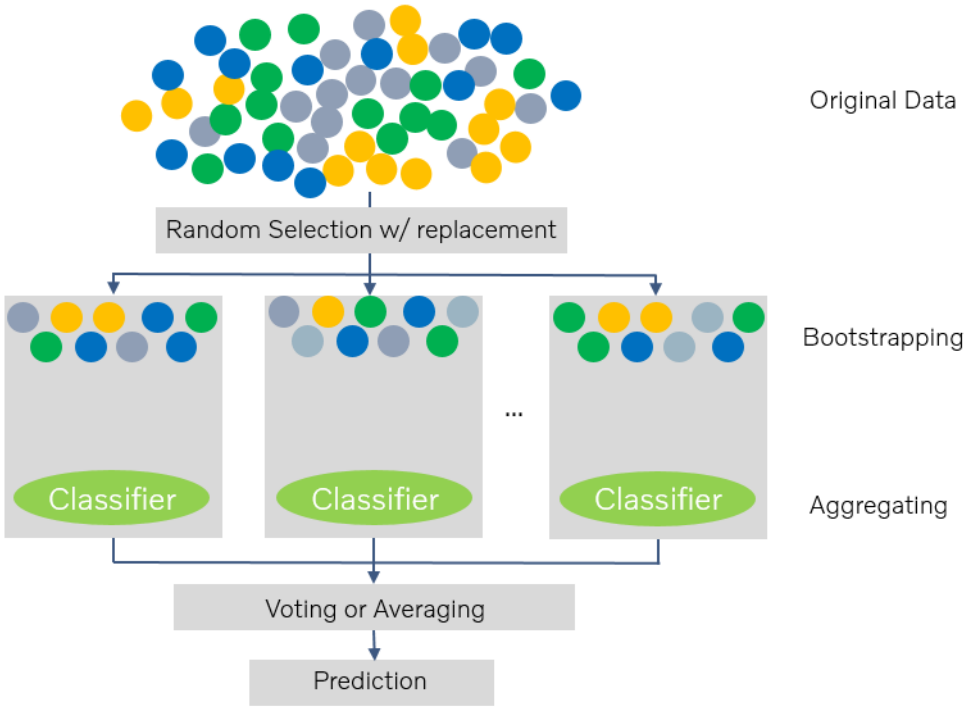


Figure 2.8: The Bagging algorithm for classification

Algorithm 1: Bagging Method [60]

Input: Training set S, classifier τ, number of bootstrap samples N .

for $i = 1...N$ **do**
  S= Bootstrap samples from S ;
  $C_i = \tau(S')$;

$C^*(x) = \underset{y \in Y}{\text{argmax}} \sum_{i, C_i = Y} 1$;

Output: Classifier C^*

2.2.3.2 Decision Trees

The Decision Tree classifier [61] is a widely used supervised classification algorithm for many types of machine learning problems, and works by breaking down a complex decision-making process into a collection of simpler decisions. This is done by forming a tree-like structure, that classifies the data by tracing the path from root to leaf. The attribute with the largest information gain is chosen as the parent node, and the subsequent attributes are assigned to the child nodes [62]. This is represented in Figure 2.9. Decision Trees have therefore the benefit of providing a solution that is very easy to interpret, compared to machine learning models with a more black-box alike approach. Each branch in the tree represents the outcome of the test made at each level in the tree.

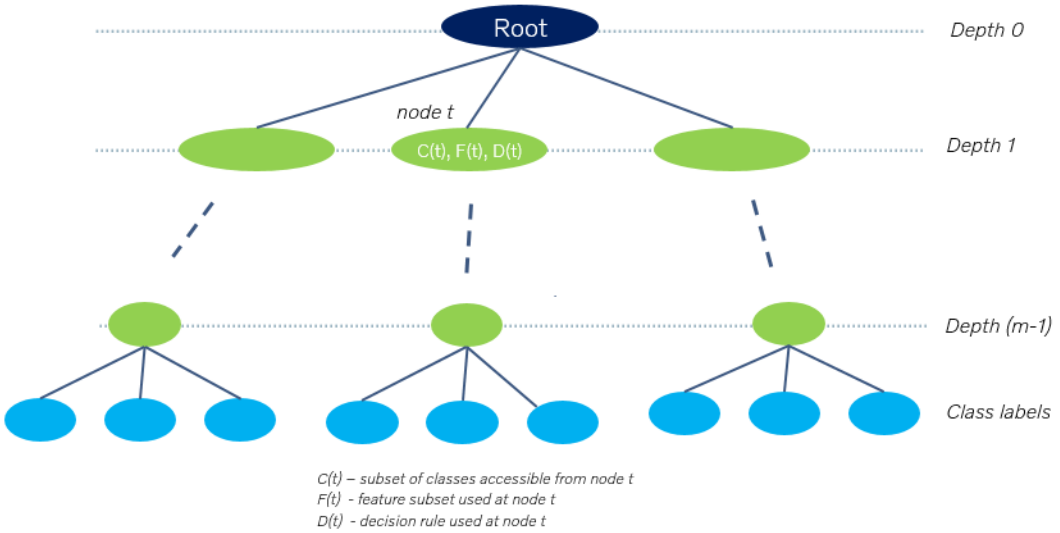


Figure 2.9: Decision Tree Algorithm

Decision Tree Classification is a very fast algorithm for both training and prediction, but is very sensitive to small perturbations of the input data. Therefore, it is often implemented by combining several Decision Trees with an ensemble method, such as Boosting [59] or Bagging [60].

2.2.3.3 Random Forest (RF)

Random Forest Algorithm [20] is one of the most commonly used supervised classification algorithms today, and works by feeding several different Decision Trees with the same input data and then combining their results by voting. This approach is very similar as the Bagging method [60] using a decision tree as the underlying classifier. However, combining predictions from multiple models by ensembeling works better if the predictions from the sub-models are uncorrelated. Therefore, Random Forest classifiers are an improvement over bagged decision trees, since the sub-trees are learned so that the resulting predictions from all of the sub-trees have less correlation. Decision trees can otherwise have a lot of similarities, and therefore

contain high correlation in their predictions, even with the use of bagging. Random Forest classifiers overcome this problems and are therefore very powerful. Moreover, they have the benefit of being very fast to train compared to other classification techniques, but are quite slow to execute once trained. A schematic overview of the algorithm is shown in Figure 2.10.

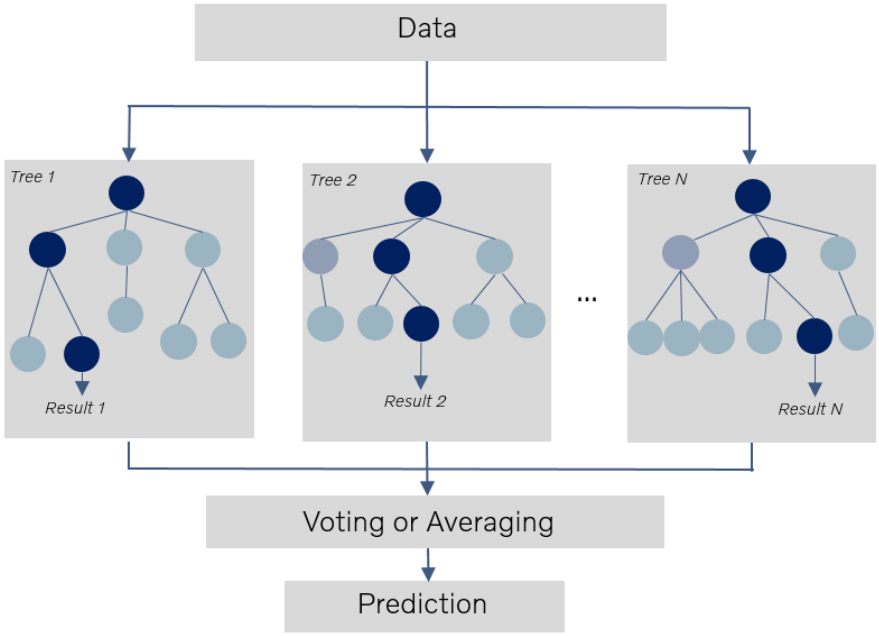


Figure 2.10: Random Forest Algorithm

2.2.3.4 Naïve Bayes (NB)

The Naïve Bayes (NB) classifier is a simple, supervised classifier that classifies data based on probabilities of events [63]. The method has the advantage of requiring less training data and training time compared to other classification methods, and is commonly applied to text classification problems in combination with a feature extraction method such as TF or TF-IDF [16]. The algorithm is based on Bayes Theorem [64], which can be described according to (2.20), saying that the probability of event A given B can be expressed using the factors of the probability of B given A, the probability of A, and the probability of B in combination as follows:

$$P(A|B) = \frac{P(A)P(B|A)}{P(B)} \tag{2.20}$$

Zhang and Gao [65] have used Bayes Theorem for text classification according to the following method. The text document to be classified is denoted as the vector $\boldsymbol{D} = \{d_1....d_n\}$, where each element d_i corresponds to a word in the document. The set of possible predefined classes that the document is going to be assigned to is denoted $\boldsymbol{C} = \{c_1....c_k\}$. Applying (2.20) to this document classification case results

in the following equation:

$$P(c_j|\boldsymbol{D}) = \frac{P(c_j)P(\boldsymbol{D}|c_j)}{P(\boldsymbol{D})} \tag{2.21}$$

Each component of (2.21) can be described as follows. $P(c_j)$ is prior information of the appearing probability of a class c_j, $P(\boldsymbol{D})$ can be retrieved from observations by knowledge from the text to be classified, and $P(\boldsymbol{D}|c_j)$ can similarly be calculated form the distribution probability of document $\boldsymbol{D}$ in classes space according to

$$P(\boldsymbol{D}|c_j) = \prod_i P(d_i|c_j) \tag{2.22}$$

Combining (2.21) and (2.22) gives the following probability expression:

$$P(c_j|\boldsymbol{D}) = \frac{P(c_j)\prod_i P(d_i|c_j)}{P(\boldsymbol{D})} \tag{2.23}$$

The NB classifier then uses this equation to compute the probability that a given document D belongs to each possible class, and assigns the document to the class which has the highest probability. This can be expressed according to

$$C^*(D) = \underset{j}{\text{argmax}} \;\; P(c_j|\boldsymbol{D}) = \underset{j}{\text{argmax}} \frac{P(c_j)\prod_i P(d_i|c_j)}{P(\boldsymbol{D})} \tag{2.24}$$

Since the $P(\boldsymbol{D})$ factor is identical to each class, it can be discarded since it will not affect the classifier's choice. The final expression for the NB classifier of text documents is

$$C^*(D) = \underset{j}{\text{argmax}} P(c_j) \prod_i P(d_i|c_j) \tag{2.25}$$

2.2.3.5 K-Nearest Neighbor (KNN)

The K-Nearest Neighbor (KNN) algorithm is a simple, supervised classification method, commonly used for text classification problems [17]. It requires the text to be feature extracted to a numerical form, such as TF-IDF vectors. The algorithm then works by finding the distances between the unlabeled text document and a specified number of K closest examples of the training data set. The algorithm then selects the most frequent label as the category which the text document gets classified into. This is demonstrated in Figure 2.11, in which the selected category would be c2 by majority voting.

Given a test example to classify $\boldsymbol{X} = \{x_1, ..., x_m\}$ as a feature extracted text document with m features, as well as j categories $\{c_1, ...c_j\}$, the process of the KNN algorithm can be described as follows [66]. If N number of training examples are available in the training set $\mathbf{D} = \{\boldsymbol{d_1}, ...\boldsymbol{d_N}\}$ where $\boldsymbol{d_i} = \{d_{i1}..., d_{im}\}$, a similarity function $SIM()$ between the test document X and one training example d_i can be

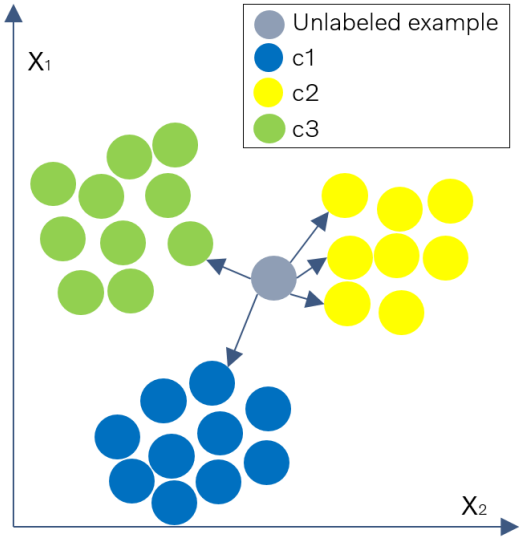


Figure 2.11: Visualization of a KNN classification example where $K = 5$, $j = 3$, and two features are present. The unlabeled text document gets classified into c_2 by majority voting.

described:

$$SIM(\boldsymbol{X}, \boldsymbol{d_i}) = \frac{\sum_{j=1}^{m} X_j \cdot d_{ij}}{\sqrt{(\sum_{j=1}^{m} X_j)^2} \cdot \sqrt{(\sum_{j=1}^{m} d_{ij})^2}} \tag{2.26}$$

The similarity between $\boldsymbol{X}$ and each training example is calculated, after which the k nearest samples with highest similarity are chosen. This set of training examples is denoted KNN. The probability of X belonging to each category is then calculated as:

$$P(\boldsymbol{X}, c_j) = \sum_{d \in \text{KNN}} SIM(\boldsymbol{X}, \boldsymbol{d_i}) \cdot y(\boldsymbol{d_i}, c_j) \tag{2.27}$$

where y is the category attribute function:

$$y(\boldsymbol{d_i}, c_j) = \begin{cases} 1, \boldsymbol{d_i} \in c_j \\ 0, \boldsymbol{d_i} \notin c_j \end{cases} \tag{2.28}$$

The test document $\boldsymbol{X}$ is then classified into the category which has the largest probability of $\boldsymbol{X}$ belonging to it, i.e:

$$c(\boldsymbol{X}) = \underset{j}{\text{argmax}}\ P(\boldsymbol{X}, c_j) \tag{2.29}$$

which in combination with (2.27) can be written as:

$$c(\boldsymbol{X}) = \underset{j}{\text{argmax}} \sum_{d \in \text{KNN}} SIM(\boldsymbol{X}, \boldsymbol{d_i}) \cdot y(\boldsymbol{d_i}, c_j) \tag{2.30}$$

The KNN algorithm has the advantages of being easy to implement, fits many kinds of feature extraction methods, and does not require training of heavy computational complexity. However, the execution of the algorithm is relatively slow since all the training data has to be processed during each document classification example. Efforts have been made to overcome this limitation. One example is Suguna and Thanushkodi [66], who use a genetic algorithm in combination with a traditional KNN. Their method succeeded to both decrease classification complexity and improve the accuracy of the KNN algorithm.

2.2.3.6 Support Vector Machine (SVM)

Support Vector Machine (SVM) [18] is a very efficient algorithm which can be applied to a huge variety of machine learning problems, and works by maximizing the margin between the class boundary and the training patterns. This is done by choosing hyper planes which divide the classes, and make the selection such that the distance from each hyper plane to the nearest data point on each side is maximized. This is illustrated for a two feature problem in Figure 2.12.

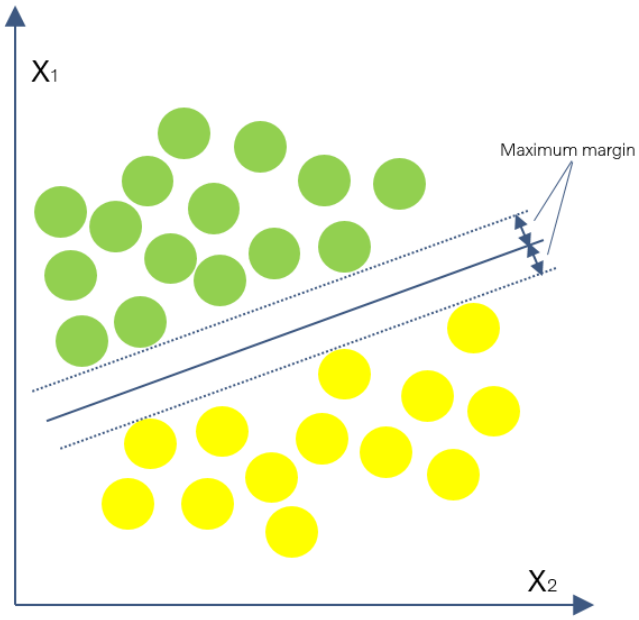


Figure 2.12: A linear decision boundary for a two dimensional Support Vector Machine, represented by a hyper plane.

The hyper planes only separate two classes and are therefore applied in a binary classification sense, however a multi-class classifier can be achieved by constructing each hyper plane with a one-vs-all strategy and then combining several hyper planes together. The training procedure of a A/B binary-class SVM algorithm works as follows. The input to the algorithm is a training set $\boldsymbol{X}$ consisting of p training

examples with corresponding labels.

$$\begin{aligned} &\boldsymbol{X} = \{(\boldsymbol{x_1}, y_1), ..., (\boldsymbol{x_p}, y_p)\} \\ &\text{where} \begin{cases} y_k = 1 & \text{if} \quad \boldsymbol{x_k} \in \text{class A} \\ y_k = -1 & \text{if} \quad \boldsymbol{x_k} \in \text{class B} \end{cases} \end{aligned} \tag{2.31}$$

In (2.31), each training example $\boldsymbol{x_i}$ is a N-dimensional vector, extracted using a feature extraction method. The goal of the classifier is to train a decision function, $D(\boldsymbol{x})$, such that an unlabeled test data example $\boldsymbol{x}$ can be classified according to

$$\begin{aligned} &\boldsymbol{x} \in A \text{ if } D(\boldsymbol{x}) > 0 \\ &\boldsymbol{x} \in B \text{ otherwise.} \end{aligned} \tag{2.32}$$

In the simplest case, this decision function consists of a hyper plane which linearly separates the two classes by maximizing the margin to the closest training data examples on each side of the decision boundary. This is done by constructing $D(\boldsymbol{x})$ according to (2.33), in which ϕ_i are pre-defined functions of $\boldsymbol{x}$, and w_i and b are the adjustable parameters of the decision function.

$$D(\boldsymbol{x}) = \sum_{i=1}^{N} w_i \phi_i(\boldsymbol{x}) + b \tag{2.33}$$

In the form of vector multiplication, this can be written as

$$D(\boldsymbol{x}) = \boldsymbol{w} \cdot \boldsymbol{\phi_i}(\boldsymbol{x}) + b \tag{2.34}$$

in which $\boldsymbol{w}$ and $\boldsymbol{\phi}$ are N dimensional vectors instead. The distance between the hyperplane defined by $D(\boldsymbol{x})$ and the training pattern $\boldsymbol{x}$ can be expressed as

$$\text{distance} = \frac{D(\boldsymbol{x})}{||\boldsymbol{w}||} \tag{2.35}$$

If the training set can be separated by the hyper plane with a margin M, all training examples will fulfill the following inequality:

$$\frac{y_k D(\boldsymbol{x_k})}{||\boldsymbol{w}||} \geq M \tag{2.36}$$

The objective of the training algorithm can then be defined as finding the parameter vector $\boldsymbol{w}$ that maximizes M, according to

$$\begin{aligned} M^* = &\max_{\boldsymbol{w}, ||\boldsymbol{w}||=1} M \\ &s.t. \quad y_k D(\boldsymbol{x_k}) \geq M, \\ &k = 1, 2 \ldots p \end{aligned} \tag{2.37}$$

The optimal bounding margin M^* is attained by the constraining examples which have the minimum margin:

$$\min_k \quad y_k D(\boldsymbol{x_k}) = M^* \tag{2.38}$$

The problem of constructing a decision function by finding a hyper plane with maximum margin to the closest training examples is therefore a max-min-problem, defined by combining (2.37) and (2.38):

$$\max_{\boldsymbol{w},||\boldsymbol{w}||=1} \quad \min_{k} \quad y_k D(\boldsymbol{x_k}) \tag{2.39}$$

However, this strategy is only applicable when the decision boundary is linear such that the data can be separated using a hyper plane. In many cases, a much more complex decision boundary is desired. Therefore, a more general approach for a SVM classifier is to construct an algorithm in which the decision boundary does not need to be linear. The most famous approach to achieve this is to project the training data from its original space $\mathcal{X}$ into a dual, higher dimensional feature space $\mathcal{F}$ [67]. Applying a similar maximum margin strategy as above then enables the possibility to construct hyper planes that are linear in the dual space, but correspond to more complex decision boundaries in the original space. The conversion $\mathcal{X} \rightarrow \mathcal{F}$ between the original feature space $\mathcal{X}$ into the dual feature space $\mathcal{F}$ is performed by using a Kernel operator, denoted K. Starting from (2.31) and (2.32), the function $D(\boldsymbol{x})$ can in the dual space be written as follows [18]:

$$D(\boldsymbol{x}) = \sum_{k=1}^{p} \alpha_k K(\boldsymbol{x_k}, \boldsymbol{x}) + b \tag{2.40}$$

The coefficients α_k are the parameters to be adjusted, and K is a predefined Kernel operator. If K satisfies Mercer's condition [68], it can be expressed as

$$\begin{aligned} K(\boldsymbol{x}, \boldsymbol{x'}) = &\sum_{i} \phi_i(\boldsymbol{x})\phi_i(\boldsymbol{x'}) \\ &\text{where } \phi : \mathcal{X} \rightarrow \mathcal{F} \end{aligned} \tag{2.41}$$

The relation between the direct parameters w_i in the original space and the dual parameters α_k can be thereafter constructed as follows:

$$w_i = \sum_{k=1}^{p} \alpha_k \phi_i(\boldsymbol{x_k}) \tag{2.42}$$

With the use of (2.42) and (2.41), the function for $D(\boldsymbol{x})$ in the dual space in (2.40) gets identical to (2.33) in the original space. Hence, the optimal hyper plane can be constructed in the dual, higher-dimensional feature space using the same maximum margin strategy as in the original feature space. By finding the best α_k parameters, a linear decision boundary in the dual space can be constructed, which in the original space corresponds to a complex decision boundary between the classes.

This method is very powerful, and SVM classifiers are one of the most accurate classifiers, especially for high-dimensional data which text data often is. The downside with SVM classifiers is that the classification is hard to interpret, meaning it is difficult to get a understanding of why a certain point has been labeled to a certain category.

2.2.3.7 Logistic Regression (LR)

Logistic Regression [15] is one of the earliest methods of classification, and works by statistical modeling of probabilities for the training examples belonging to each category. In the simplest form, it is used for binary classification in which the algorithm outputs a probability between 0 and 1. This is done with usage of the Sigmoid function $\sigma(z)$, which approaches 1 for infinitely high positive numbers and 0 for infinitely high negative numbers. The equation is defined as

$$\sigma(z) = \frac{1}{1+e^{-z}} \tag{2.43}$$

The algorithm trains a number of parameters stored in a vector $\boldsymbol{\theta}$, and the text data training set $\boldsymbol{X} \in \mathbb{R}^{nxd}$ consists of n training examples, each feature extracted to a numerical form with d features. The $\boldsymbol{\theta}$-vector gets vector multiplied with a training example $\boldsymbol{x_i}$, which then gets classified according to the resulting probability $\sigma(\boldsymbol{\theta^T x_i})$ being higher or lower than 0.5. This corresponds to if the value $\boldsymbol{\theta^T x_i}$ is negative or positive, illustrated in Figure 2.13.

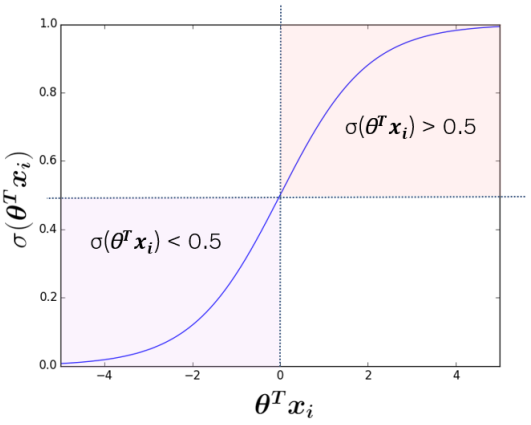


Figure 2.13: Sigmoid function illustrated for a binary classification case of $\boldsymbol{\theta^T x_i}$

If the label 0 or 1 for each training example is denoted y_i, the probability of a training example belonging to a certain class can be derived as follows:

$$\begin{aligned} p(y_i = 1|\boldsymbol{x_i}, \boldsymbol{\theta}) &= \sigma(\boldsymbol{\theta^T x_i}) \\ p(y_i = 0|\boldsymbol{x_i}, \boldsymbol{\theta}) &= 1 - \sigma(\boldsymbol{\theta^T x_i}) \end{aligned} \tag{2.44}$$

These can be put together in a more compact form as follows:

$$p(y_i|\boldsymbol{x_i}, \boldsymbol{\theta}) = \sigma(\boldsymbol{\theta^T x_i})^{y_i}(1 - \sigma(\boldsymbol{\theta^T x_i}))^{(1-y_i)} \tag{2.45}$$

The goal of Logistic Regression is to train the $\boldsymbol{\theta}$-parameters from the probability of the class labels $\boldsymbol{Y}$ being 0 or 1, given the training data $\boldsymbol{X}$. This is done using

the Bernoulli mixture models function [69], which using (2.45) defines a likelihood of the $\boldsymbol{\theta}$-parameters as follows:

$$\begin{aligned} L(\boldsymbol{\theta}) = \quad & \prod_{i=1}^{n} p(y_i|\boldsymbol{x_i}, \boldsymbol{\theta}) = \\ & \prod_{i=1}^{n} \sigma(\boldsymbol{\theta^T x_i})^{y_i} (1 - \sigma(\boldsymbol{\theta^T x_i}))^{(1-y_i)} \end{aligned} \tag{2.46}$$

When maximizing a certain expression or likelihood, the same result is obtained as maximizing the log likelihood. Together with the concept of logarithmic rules which allows the log of products to be written as the sum of each log, the log likelihood in logistic regression can be derived as follows:

$$\begin{aligned} \log(L(\boldsymbol{\theta})) = \quad & \log(\prod_{i=1}^{n} p(y_i|\boldsymbol{x_i}, \boldsymbol{\theta})) = \\ & \sum_{i=1}^{n} y_i \log(\sigma(\boldsymbol{\theta^T x_i}))) + (1 - y_i) \log(1 - \sigma(\boldsymbol{\theta^T x_i}))) \end{aligned} \tag{2.47}$$

From (2.47), a cost function $J(\boldsymbol{\theta})$ for logistic regression can be obtained, which is proportional to the inverse of the log likelihood:

$$J(\boldsymbol{\theta}) = \sum_{i=1}^{n} -y_i \log(\sigma(\boldsymbol{\theta^T x_i}))) - (1 - y_i) \log(1 - \sigma(\boldsymbol{\theta^T x_i}))) \tag{2.48}$$

The aim of the training process of a logistic regression based classification algorithm is to find the parameters $\boldsymbol{\theta}$ such that the cost function $J(\boldsymbol{\theta})$ defined in (2.48) is minimized. This is commonly done using gradient descent.

If more than two classes are present, the above strategy can also be extended to a multi-class classifier. This is called Multinominal Logistic Regression [69] which instead of describing the probabilities using the Sigmoid function uses a Softmax. To describe the probability that a training example $\boldsymbol{x_k}$ belongs to a certain class i, the softmax function can be written as (2.49), in which the label for class k is represented as a one-hot encoded vector $\boldsymbol{y_k}$, equal to one on the ith position if the class index is i:

$$p(y_{k,i} = 1|\boldsymbol{x_k}, \boldsymbol{\theta}) = \frac{e^{\boldsymbol{\theta_i^T x_k}}}{\sum_{j=1}^{m} e^{\boldsymbol{\theta_j^T x_k}}} \quad \text{for } i \in \{1, ...m\} \tag{2.49}$$

If $m = 2$, the above equation corresponds to the binary logistic regression, and for $m > 2$ it is used as multinominal logistic regression. Since the equation entails a normalization condition, the sum of all probabilities are always equal to 1:

$$\sum_{i=1}^{m} p(y_{k,i} = 1|\boldsymbol{x_k}, \boldsymbol{\theta}) = 1 \tag{2.50}$$

The log likelihood for logistic regression in (2.47) can be extended to the multinominal case:

$$\begin{aligned} \log(L(\boldsymbol{\theta})) = & \sum_{j=1}^{n} \log\, p(\boldsymbol{y_j}|\boldsymbol{x_j}, \boldsymbol{\theta}) = \\ & \sum_{j=1}^{n} \left[\sum_{i=1}^{m} y_{j,i}\ \boldsymbol{\theta_i^T}\ \boldsymbol{x_j} - \log\ \sum_{i=1}^{m} e^{\boldsymbol{w_i^T x_j}} \right] \end{aligned} \tag{2.51}$$

Similar to the binary logistic regression case, the $\boldsymbol{\theta}$-parameters are then trained to maximize the likelihood of the training data belonging to its classes.

2.2.3.8 Deep Learning

Deep learning is a topic within AI which recently have achieved state-of-the-art results in many areas, including Natural Language Processing. Many different types of Deep Learning based classification techniques exist, of which some of the most common will be described further.

2.2.3.8.1 DNN: Deep Neural Network (DNN) [21] are based on the concept of logistic regression by connecting a large number of logistic regression alike units together. Each unit is called a neuron, and these are stacked together in parallel, forming layers. A DNN is then designed as connecting these computational layers in series, where each layer's output is the next layer's input . The first layer, called input layer, takes the feature extracted text vector $\boldsymbol{x}$ as input, while the last layer, called the output layer, outputs a number of probabilities of the input vector belonging to each of the possible classes, $\boldsymbol{y}$. The class with the highest probability is then selected as the predicted class. A schematic representation of a general neural network is represented in Figure 2.14. Every neuron consists of a set of mathematical equations, and the concept of a network is only a visual representation of how the algorithm is built. Therefore, the term Artificial Neural Network is commonly used.
Each neuron in a DNN takes all the neuron in the previous layer's output as input, which is then multiplied with weights, $\boldsymbol{\theta}$. These are then added, together with a bias term. The resulting scalar number is thereafter put through an activation function, such as the Sigmoid function described above in equation 2.43, whose output corresponds to the final output of the neuron. A closer look at the algorithm that each neuron performs is given in Figure 2.15.

Several activation functions exist. If no activation function is used, all neurons would be linear, and the complete network could be simplified to a linear mapping function between $\boldsymbol{x}$ and $\boldsymbol{y}$, which would entail severe limitations in the possible complexity of the network. Therefore, by introducing non-linear activation functions, the complete network can be simulated as a non-linear function f, able to define a complex mapping $\boldsymbol{y} = f(\boldsymbol{x})$ between the input and output data, which without the usage of a neural network would be difficult to obtain. The three most common activation functions are the Sigmoid function [2.52], the Hyperbolic Tangent Function (tanh)

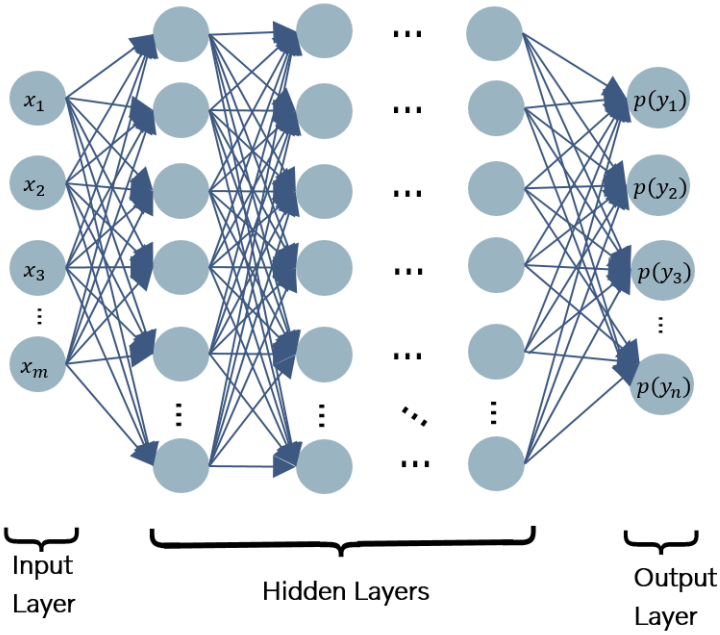


Figure 2.14: Schematic view of a Deep Neural Network

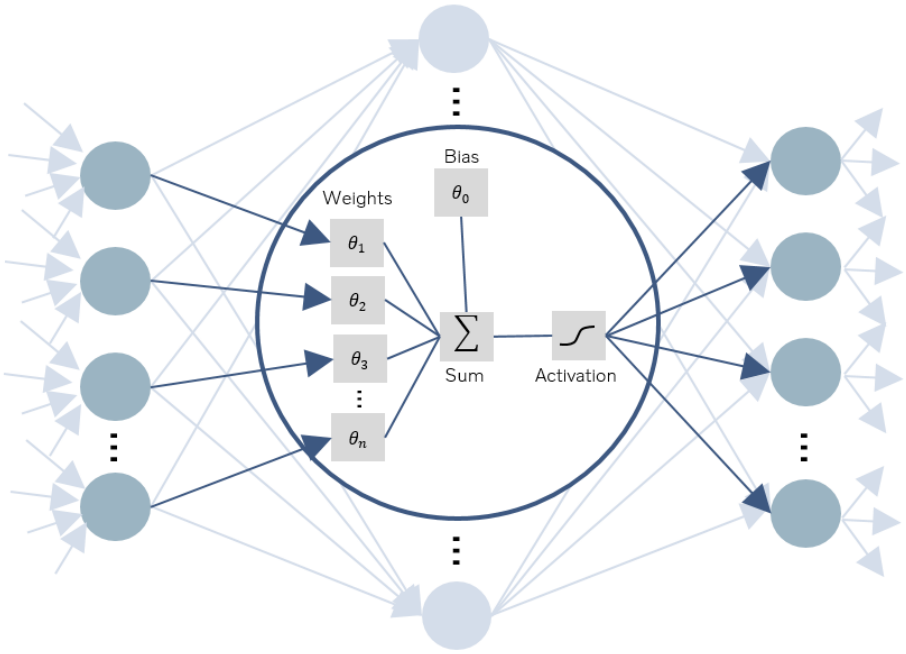


Figure 2.15: The algorithm of a single neuron in a Neural Network

[2.53] and the Rectified Linear Unit (ReLU) [2.54]. Furthermore, the output layer often uses a Softmax function [2.55], since it provides a good representation of the probabilities of each class in a multi-class scenario.

$$\text{sigmoid}(z) = \sigma(z) = \frac{1}{1+e^{-z}} \tag{2.52}$$

$$\tanh(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}} \tag{2.53}$$

$$\text{relu}(z) = \max(0, z) \tag{2.54}$$

$$\text{softmax}(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{m} e^{z_j}} \quad \forall\ i \in \{1, ...m\} \tag{2.55}$$

By observing a large number of labeled training example pairs $(\boldsymbol{x}, \boldsymbol{y})$, $\boldsymbol{x} \in \boldsymbol{X}$, $\boldsymbol{y} \in \boldsymbol{Y}$, the network trains by tweaking and tuning the weights and biases such that the error between the predicted class labels $\hat{\boldsymbol{y}}$ and the actual class labels $\boldsymbol{y}$ for all training examples is minimized. Looping through all the training examples available, this process is done in two steps. In the first step, the algorithm in each neuron is executed, from the input layer to the output layer, such that each neuron's activation value can be calculated. Since this is done forward in the network, the step is called forward propagation. To keep track of the performance of the network, a cost function $J(\boldsymbol{\theta}, \boldsymbol{x_i}, \boldsymbol{y_i})$ is defined. In the second step, the value of the derivative of J with respect to $\boldsymbol{\theta}$ at the point of the neuron's activation value, $\Delta_\theta J(\boldsymbol{\theta}, \boldsymbol{x_i}, \boldsymbol{y_i})$ is calculated for each neuron. Since this is done backwards in the network, from the output layer to the input layer, the step is called backpropagation. By performing these two steps, the learning algorithm knows in what direction the $\boldsymbol{\theta}$- parameters should be changed in order to decrease the cost function. Using some of the optimization techniques described in 2.2.3.9, each individual component of the $\boldsymbol{\theta}$-parameters is optimized such that the defined cost function J is minimized. This is the actual training of the network, and defines the learning process of finding the best function mapping $\boldsymbol{X} \rightarrow \boldsymbol{Y}$ of the training data.

2.2.3.8.2 Recurrent Neural Network (RNN): Another commonly used network structure within deep learning is Recurrent Neural Networks (RNNs) [21]. These have the unique property of being able to remember previous data points of a sequence, by letting the layer outputs reenter as input to the layer. This is advantageous for analyzing time-series data, such as natural language. Denoting x_t as the state at time t and $\boldsymbol{u_t}$ as the input at step t, the general RNN formula can be expressed as

$$x_t = F(x_{t-1}, \boldsymbol{u_t}, \theta) \tag{2.56}$$

Equation (2.56) can be extended by introducing a recurrent matrix weight $\boldsymbol{W_{rec}}$, input weights $\boldsymbol{W_{in}}$, element-wise product function E, as well a the bias b:

$$x_t = E(\boldsymbol{W_{rec}}, x_{t-1}) + \boldsymbol{W_{in}}\boldsymbol{u_t} + \boldsymbol{b} \tag{2.57}$$

Basic RNNs entail several limitations, of which the most common are problems with vanishing or exploding gradient during the back propagation step of training the network. To overcome these problems, a special type of network cell called Long Short-Term Memory (LSTM) [70] is introduced, which preserves the network's long term dependency in a more effective way compared to a basic RNN. A simplified version of LSTMs is Gated Recurrent Units (GRU) [71], which also is common to use as RNN components. The architecture of a RNN with either LSTM or GRU cells is illustrated in Figure 2.16.

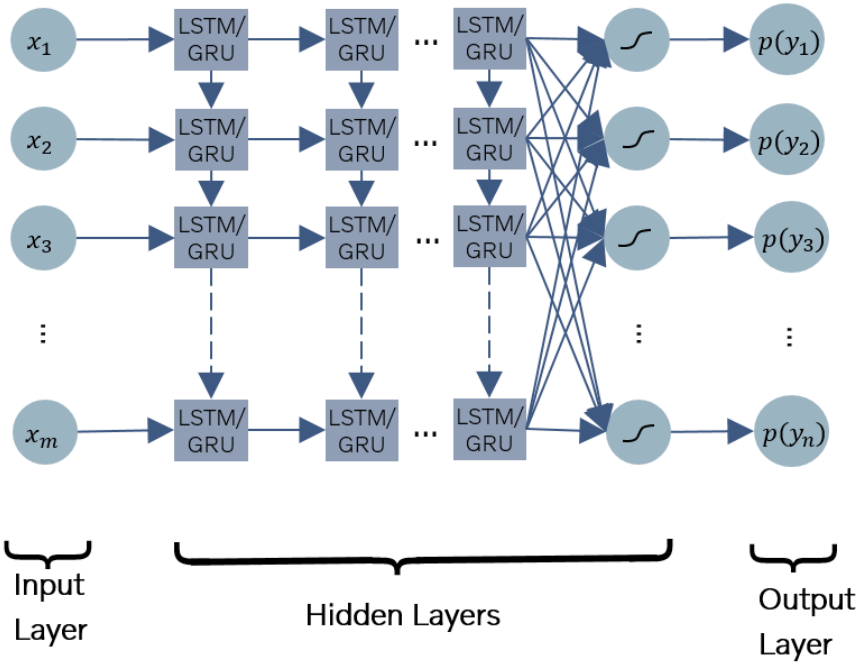


Figure 2.16: The RNN architecture

The LSTM cells work by using multiple gates to regulate the amount of information allowed into each node state. A step-by-step explanation is provided below, where the sigmoid- and tanh functions correspond to (2.52) and (2.53) above. The indices i, c, f and o refer to input, cell memory, forget gate and output gates respectively, while b represents a bias vector, W a weight matrix, and x_t the input to the memory cell at time t.

1. **Candidate Memory Cell:** the hidden state of a RNN.
$$\tilde{C}_t = \tanh(W_c[x_t + h_{t-1}] + b_c) \tag{2.58}$$
2. **Input Gate:** also called update gate, decides if the cell state should be updated with the candidate state or not, i.e., how much of the past state should matter now.
$$i_t = \sigma(W_i[x_t + h_{t-1}] + b_i) \tag{2.59}$$
3. **Forget Gate:** controls how much is kept from the previous cell state, and forgets the rest.
$$f_t = \sigma(W_f[x_t + h_{t-1}] + b_f) \tag{2.60}$$

4. **New Memory Cell:** removes some content from last cell state, and writes some new cell content.

$$C_t = i_t * \tilde{C}_t + f_t * C_{t-1} \tag{2.61}$$

5. **Output Gate:** controls which part of the cell is output to the hidden state and determines what the next hidden state will be.

$$o_t = \sigma(W_o[x_t + h_{t-1}]) + b_o \tag{2.62}$$

6. **Activation:** passes the activation to the next cell.

$$h_t = o_t * \tanh(C_t) \tag{2.63}$$

A visual representation of a single LSTM cell is shown in Figure 2.17.

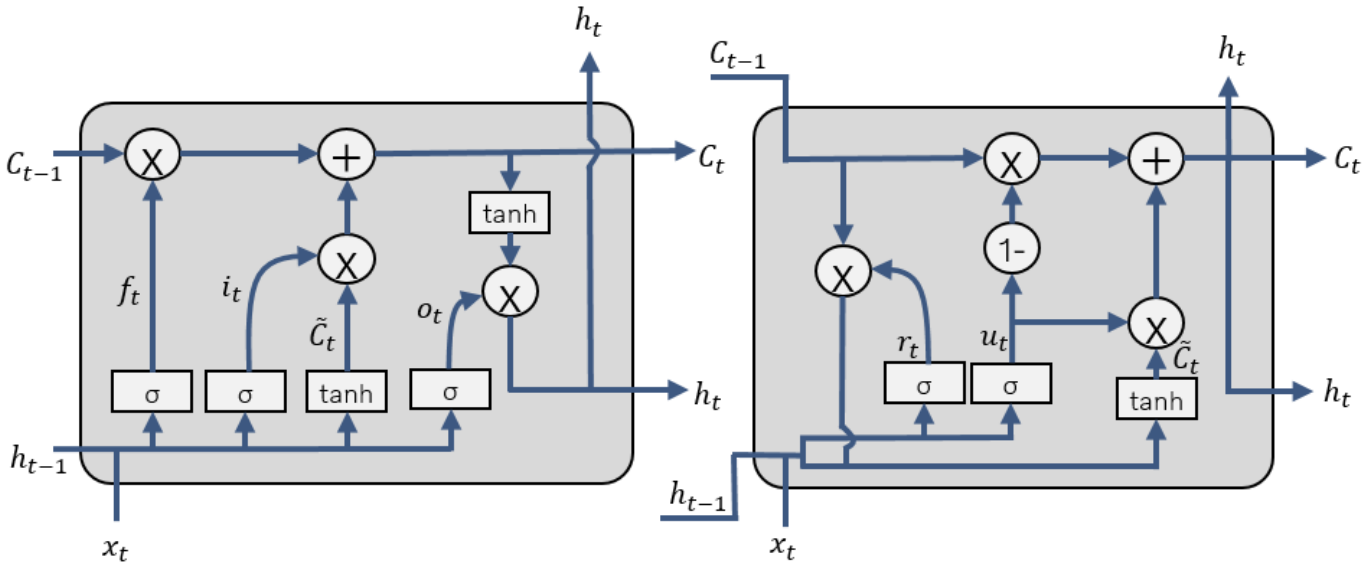


Figure 2.17: Schematic view of a LSTM cell (left) and a GRU cell (right)

The GRU cell is a simplified variant of the LSTM architecture, containing two gates that are reset and update gates, instead of the three input, output and forget gates. It therefore uses less memory, execute faster, and train faster than LSTMs, however with the tradeoff of lower accuracy. A visual representation of a single GRU cell is shown in Figure 2.17, and it works as follows.

1. **Candidate Memory Cell:** the hidden state of a RNN.

$$\tilde{C}_t = \tanh(W_c[x_t + r_t h_{t-1}] + b_c) \tag{2.64}$$

2. **Update Gate:** decides if the cell state should be updated with the candidate state or not, i.e. how much of the past state should matter now.

$$u_t = \sigma(W_u[x_t + h_{t-1}] + b_u) \tag{2.65}$$

3. **Reset Gate:** decides whether the previous cell state is important or not, i.e. how much information is irrelevant in the future.

$$r_t = \sigma(W_r[x_t + h_{t-1}] + b_r) \tag{2.66}$$

4. **New Memory Cell:** removes some content from last cell state, and writes some new cell content.

$$C_t = u_t * \tilde{C}_t + (1 - u_t)C_{t-1} \tag{2.67}$$

5. **Activation/Output Vector:** passes the activation to the next cell.

$$h_t = C_t \tag{2.68}$$

2.2.3.8.3 Convolutional Neural Network (CNN): Convolutional Neural Network (CNN) [72] is a powerful deep learning architecture which was primarily developed for image processing, but has also been used successfully within NLP [73]. Unlike regular DNNs, CNNs entail at least one convolutional layer rather than only fully connected ones. Convolutional layers are named from the mathematical convolution operation, which works as follows [26]. A 2-dimensional matrix is given as input, for example pixel intensity for a gray scaled image or stacked row vectors of the feature extracted words in a text document. A chosen number of filters are then created, each as a 2 dimensional matrix. The content of these matrices are the trainable parameters of the layer, which the CNN algorithm will optimize. A window is then selected with the same size as the filter, which is transitioned over the input image with a certain number of pixels in each transition. For each window frame, a convolution operation is made between the observed window and the filter, which in practice means to sum the element wise multiplication of the filter parameters and the numbers in the window. This is illustrated in Figure 2.18. The resulting sum is the position in the output layer corresponding to the filter and window position. The size of the output layer is therefore purely dependent on the selection of filter size and the number of pixel steps that the window is moved between the convolution operations.

If more than one filter is used, the output will be 3-dimensional, where the third dimension corresponds to the number of filters used. Because of this, the size of the CNN layers when connecting many convolution layers together can become huge, thus entailing large computational complexity. To overcome this problem while still preserving important features, pooling layers are often introduced between the convolution layers. One example of pooling is shown in Figure 2.19 where the operation is of the form *max pool*, which simply works by taking the maximum value in each window. Pooling layers therefore lack trainable parameters, but do entail a number of hyper parameter selections such as window size and type of pooling operation used.

In a complete CNN architecture, convolutional layers are often stacked after each other, with pooling layers between. When the dimensions have been reduced enough, the results of the last pooling layer are concatenated together to a 1D array, after

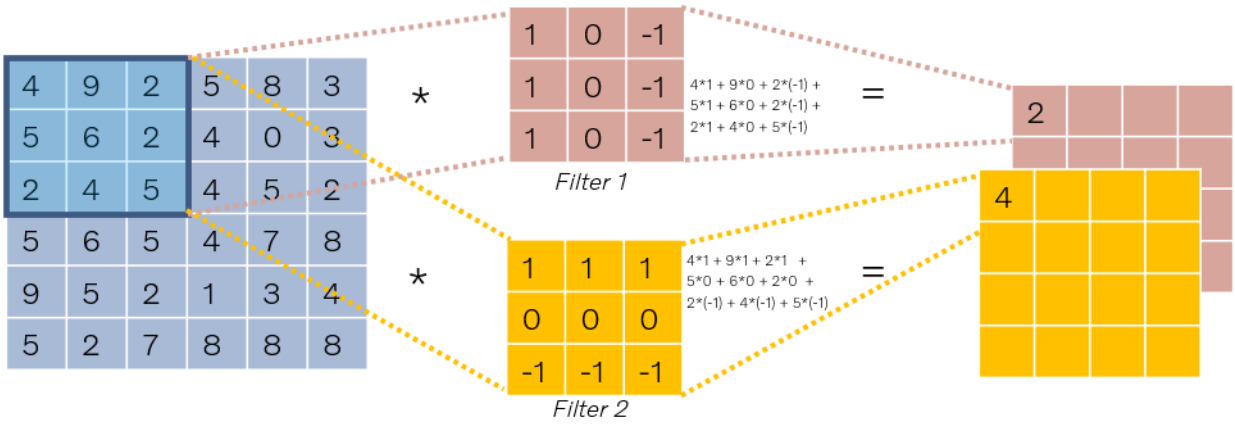


Figure 2.18: The convolution operation in a CNN

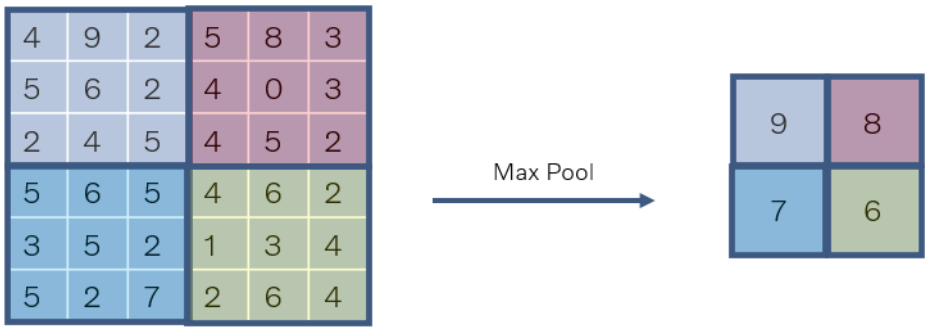


Figure 2.19: The pooling operation in a CNN

which a number of fully connected layers are used to produce the final outcome of the network. A schematic view of a CNN algorithm for text classification is illustrated in Figure 2.20.

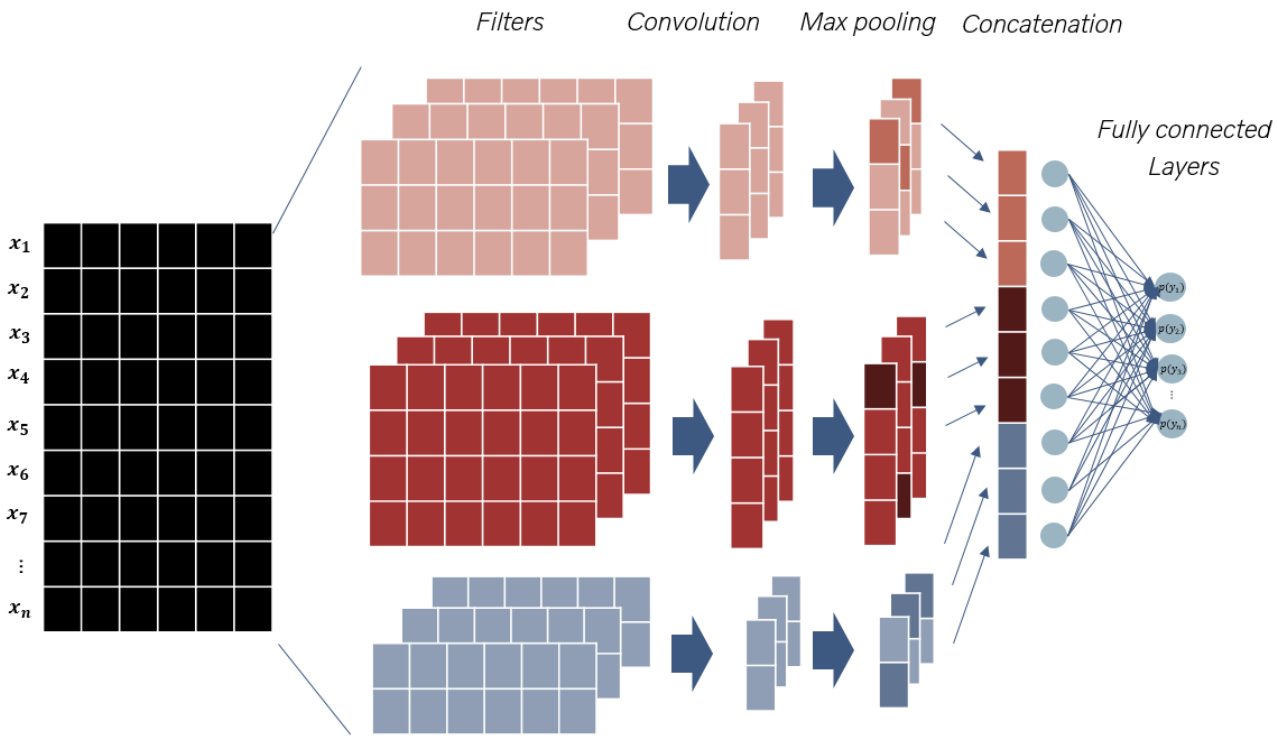


Figure 2.20: A schematic view of a CNN for text classification

2.2.3.9 Optimization Techniques

Many machine learning models, especially the deep learning based ones, are trained by optimizing the model parameters such that a constructed loss- or cost function is minimized. This optimization process is crucial for ensuring the best fit between the predicted and actual outputs. Many optimization techniques exist, of which the most common are presented below.

2.2.3.9.1 Stochastic Gradient Descent (SGD): One of the most basic form of optimization is Stochastic Gradient Descent (SGD) [74], which given a cost function J finds the gradient with respect to the parameters to be optimized, $\boldsymbol{\theta}$. The negative gradient is the direction of which a step is taken which eventually gets closer to the optimum. How big the step is, is defined by the so called learning rate, α. A too low learning rate makes the algorithm reach the optimum slower, while a learning rate that is too high could result in a overshoot of the optimum, thus diverge from the best solution. The equation for one iteration of a SGD algorithm is

$$\boldsymbol{\theta} := \boldsymbol{\theta} - \alpha \nabla_{\theta} J(\boldsymbol{\theta}) \tag{2.69}$$

The word "stochastic" means that the system is linked with a random probability. In SGD, this comes from the fact that the samples are selected randomly - the parameter update is performed for every single training example $\boldsymbol{x_i}, \boldsymbol{y_i}$. The opposite is called Batch Gradient Descent (BGD), which calculates the loss function gradient for the whole training set before the update step is performed. In this way, every step towards the minimum loss function point gets more precise, resulting in a more straight optimization path towards the optimum point compared to SGD. This is illustrated in Figure 2.21. However, the smooth and precise optimization path in path in BGD comes with the trade off of being very computationally expensive, since each and every training example has to be used in order to perform one single parameter update iteration. Therefore, a common approach is to use so called Mini-batch Gradient Descent (MBGD), which combines the fast update speed of SGD with the precision in the update direction which BGD entails by dividing the training set into mini-batches. The parameter update is then done for the entire mini-batch, for which a size equal to 1 corresponds to SGD and a size equal to the training set size corresponds to BGD . Hence, both the learning rate α and the mini-batch size are two important hyper parameters which needs to be tuned in order for the gradient descent algorithm to work well.

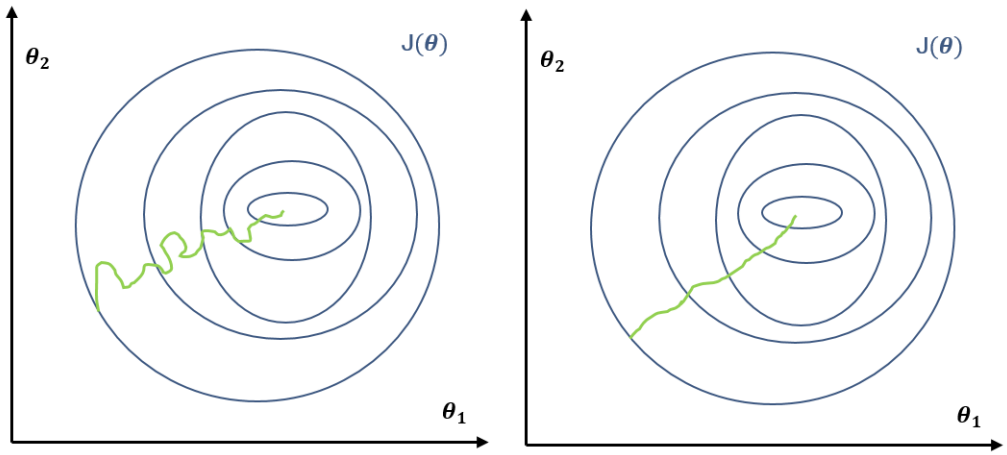


Figure 2.21: A contour plot of the cost $J(\boldsymbol{\theta})$ as a function of a two-dimensional feature vector θ_1 and θ_2 when optimized by Stochastic Gradient Descent (left) and Batch Gradient Descent (right).

A common improvement of the gradient descent algorithm is to use gradient descent with momentum. A standard gradient descent algorithm often takes larger steps in one direction and smaller steps in another, which causes an oscillating behavior and slows down the convergence. By using momentum, this unequal step size for different directions gets reduced, which more efficiently optimizes the parameter towards the minimum. The equation for one iteration of Gradient Descend with momentum factor γ is

$$\boldsymbol{\theta} := \boldsymbol{\theta} + \gamma \Delta \boldsymbol{\theta} - \alpha \nabla_\theta J(\boldsymbol{\theta}) \tag{2.70}$$

2.2.3.9.2 Root Mean Square Propagation (RMSprop): Another common optimization technique is RMSprop [75], which similarly to Gradient Descent restricts the oscillations in the iteration process so that the update steps become more efficient in the right directions towards the minimum. This is done with adaption of the learning rate for each individual parameter, by dividing the learning rate for a weight with the running average of the magnitude of recent gradient for that weight. The equations can be derived in two steps as follows. Firstly, the moving average of squared gradients $\boldsymbol{v}$ is calculated in (2.71). Secondly, the parameter weights $\boldsymbol{\theta}$ are individually updated in (2.72). Here, ϵ is a smoothing term to avoid dividing by zero.

$$\boldsymbol{v} := \gamma \boldsymbol{v} + (1 - \gamma)(\nabla_\theta J(\boldsymbol{\theta}))^2 \tag{2.71}$$

$$\boldsymbol{\theta} := \boldsymbol{\theta} - \frac{\alpha}{\sqrt{\boldsymbol{v} + \epsilon}} \nabla_\theta J(\boldsymbol{\theta}) \tag{2.72}$$

2.2.3.9.3 Adaptive Gradient Algorithm (Adagrad): Another Gradient Descent based optimizer is Adagrad [76], which similarly to RMSprop adapts the learning rate to the individual parameters. Sparser parameters, i.e. parameters that have the value 0 in most entries, get increased learning rates, and less sparse parameters get decreased learning rate. Adagrad is therefore particularly powerful when dealing with sparse data, since the convergence is improved compared to traditional stochastic gradient descent. The algorithm works as follows. Firstly, the partial derivative of the objective function with respect to the parameter θ_i is denoted as g_i:

$$g_i = \nabla_\theta J(\theta_i) \tag{2.73}$$

A diagonal matrix $\boldsymbol{G}$ is then defined as follows:

$$\boldsymbol{G} = \sum_{i=1}^{t} g_i g_i^T \tag{2.74}$$

where each diagonal element $G_{j,j}$ is the sum of the squares of the gradients:

$$G_{j,j} = \sum_{i=1}^{t} g_{i,j}^2 \tag{2.75}$$

The update rule is finally defined:

$$\theta_i := \theta_i - \frac{\alpha}{\sqrt{G_{i,i} + \epsilon}} \cdot g_i \tag{2.76}$$

Equation (2.76) can be vectorized by performing a matrix-vector product operation $\odot$ between G and g:

$$\boldsymbol{\theta} := \boldsymbol{\theta} - \frac{\alpha}{\sqrt{\boldsymbol{G} + \epsilon}} \odot \boldsymbol{g} \tag{2.77}$$

2.2.3.9.4 Adaptive Moment Estimation (Adam): Another extension of Stochastic Gradient Descend is the Adam Optimizer [77], which is a very popular technique since it combines the advantages of RMSprop, which works well in on-line and non-stationary settings, and AdaGrad, which works well with sparse gradients. Similarly to RMSprop, the Adam method computes individual learning rates for different parameters, and bases these from estimation of the first and second moments v and m of the gradients, however with relatively low memory usage. The algorithm is explained in (2.78) - (2.82) below:

$$m := \beta_1 m + (1 - \beta_1)\nabla_{\boldsymbol{\theta}} J(\theta_i) \tag{2.78}$$

$$v := \beta_2 v + (1 - \beta_2)(\nabla_{\boldsymbol{\theta}} J(\theta_i))^2 \tag{2.79}$$

$$\hat{m} = \frac{m}{1 - \beta_1} \tag{2.80}$$

$$\hat{v} = \frac{v}{1 - \beta_2} \tag{2.81}$$

$$\theta_i := \theta_i - \alpha \frac{\hat{m}}{\sqrt{\hat{v}} + \epsilon} \tag{2.82}$$

2.2.3.9.5 Adadelta: The Adadelta algorithm [78] is a more robust extension of Adagrad, which adapts learning rates based on a moving window of gradient updates instead of accumulating all past gradients. This method tends to reduce the Adagrad's drawbacks of vanishing gradient problems, as well as the need for a manually selected global learning rate. It works as follows:

$$g := \gamma g + (1 - \gamma)\nabla_{\theta} J(\theta)^2 \tag{2.83}$$

$$x := \gamma x + (1 - \gamma)v^2 \tag{2.84}$$

$$v = -\frac{\sqrt{x + \epsilon}\delta J(\theta)}{\sqrt{g + \epsilon}} \tag{2.85}$$

2.2.4 Evaluation

Regarding the comparison of previously described algorithms and methods, well defined measuring techniques that can be applied to all models are preferable [2]. However, Yang [79] emphasizes the issue of performance inconsistency between studies, which is caused by the lack of a standard data collection. Furthermore, even if the same data collection might be used in two different studies, the partitioning of training versus test data can introduce inconsistency as well. Hence, there is a desire for a universal data collection to be shared among researchers within the text classification branch in order to distinguish the relative method advantages and drawbacks. Another important aspect of the evaluation process is knowledge of what

an evaluation metric actually measures [22]. Understanding the underlying calculations of various evaluation metrics can help in both deciding which method to use, but also ease the interpretation of how classifiers in other reports perform according to different evaluation measures. There are both numeric metrics and graphical representations of performance, which all are based on a confusion matrix similar to that in Figure 2.22. The matrix contains the four quantities: true positive (TP), false positive (FP), false negative (FN), and true negative (TN). These variables are merely sums over the relations between predictions and targets, which implies that the true classification of the test data must be known. The relevance of each specific quantity is determined by the problem at hand. For instance, if a patient is tested for a disease to potentially receive a safe, cheap, and effective treatment, the FNs are far more important that the FPs due to the risk of not offering treatment to sick patients. On the contrary, if a new treatment is supposed to undergo a test phase, the FPs are crucial for avoiding giving medical care to unaffected people.

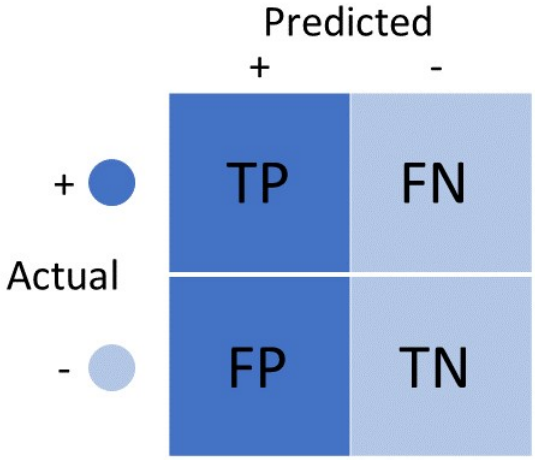


Figure 2.22: Confusion matrix showing the predicted classifications against the known data. The dark and light blue squares represent positive and negative predictions respectively. On the left side of the matrix there are two circles, where the dark symbolizes the data points known to be positive and the light circle symbolizes the negative ones.

Some of the more basic evaluation metrics are accuracy (2.86), recall (2.87), specificity (2.88), and precision (2.89). Accuracy measures the fraction of predictions that are true. Although it can be used for its straightforward interpretation, a high accuracy does not always represent a decent classifier due to the lack of information on the ratio of false predictions that are either positive or negative. The issue of FN predictions can be evaluated through the recall metric, which quantifies the fraction of known positives that are predicted correctly. However, the recall metric discards all information regarding TNs and FPs, which means that a classifier that predicts all data points as positive will have a high recall value. Vice versa, specificity measures the fraction of actual negatives that have been accurately predicted. Thus, it suffers from similar drawbacks as recall by omitting FNs and TPs. Precision captures both TPs and FPs to find the fraction of predicted positives that are accurate,

while not considering any type of predicted negatives. One of the drawbacks of this metric is that classifiers that predict only one data point as positive, given that the point has the highest probability of being positive, will yield a perfect precision score even if they miss all the other true positive data points. To avoid these typical shortcomings, and to summarize the confusion matrix more completely, (2.86) - (2.89) can be aggregated into more complex evaluation metrics.

$$\text{accuracy} = \frac{TP + TN}{TP + FP + FN + TN} \tag{2.86}$$

$$\text{recall} = \frac{TP}{TP + FN} \tag{2.87}$$

$$\text{specificity} = \frac{TN}{FP + TN} \tag{2.88}$$

$$\text{precision} = \frac{TP}{TP + FP} \tag{2.89}$$

2.2.4.1 F_β score

One of the most used aggregated evaluation metrics is the F_β score, which is a combination of the precision and recall metrics as seen in (2.90). The value of β adjusts the balance between the two metrics. Decreasing the β value increases the importance of precision while decreasing the effect of recall, and vice versa.

$$\begin{aligned} F_\beta &= (1 + \beta^2)\frac{\text{precision} \cdot \text{recall}}{\beta^2 \cdot \text{precision} + \text{recall}} \\ &= \frac{(1 + \beta^2)TP}{(1 + \beta^2)TP + \beta^2 FN + FP} \end{aligned} \tag{2.90}$$

A common β value is 1, which gives equal weight to precision and recall. Consequently, (2.90) can be simplified to (2.91). Although the F_β score is applicable in a wide range of evaluation scenarios, it is not suitable for evaluation purposes where TNs are of importance, since it does not involve that variable.

$$\frac{2TP}{2TP + FN + FP}. \tag{2.91}$$

3
Methods

The text classification algorithm was implemented in the programming language Python [80], and divided into three major steps:

1. Pre-Processing, which takes raw text data as input and outputs processed text.
2. Feature Extraction, which inputs text data and converts it to numerical features.
3. Classification, which takes the feature extracted data as input and outputs a predicted class.

Dimensionality Reduction, which is a common step in many NLP -problem, was not included in the algorithm. The reason is that the text annotations to be classified are very short, thus entailing quite small data dimensions and fast execution time of the implemented algorithm. An extra dimensionality reduction step was therefore seen as unnecessary.

In order to create a text classifier, the different algorithms involved need to be trained. Furthermore, different combinations of available techniques need to be evaluated. As a first step, in order to know what algorithms to select, a training- and evaluation phase was carried out. This involved extracting labeled training data, perform the training of the feature extraction- and classification algorithms, and finally evaluating what combination of these that performed best. A schematic view of this training phase is shown in Figure 3.1.

The second step was then the execution phase, where the selected techniques are put together in a pipeline, which forms the actual text classification algorithm that is able to classify new text data into one of the categories. A schematic view of this pipeline is shown in Figure 3.2.

All involved techniques and how they were trained are explained below.

3.1 Data Extraction

In order to get data to train the algorithm, labeled text annotations were extracted from one of Volvo Car's databases. These labels were auto generated from Volvo Car's annotation platform. A histogram of the 100 most common auto labeled data classes is shown in Figure 3.3.

It is possible to observe that the classes in Figure 3.3 are too many and not useful for a machine learning algorithm. To more easily distinguish between different classes,

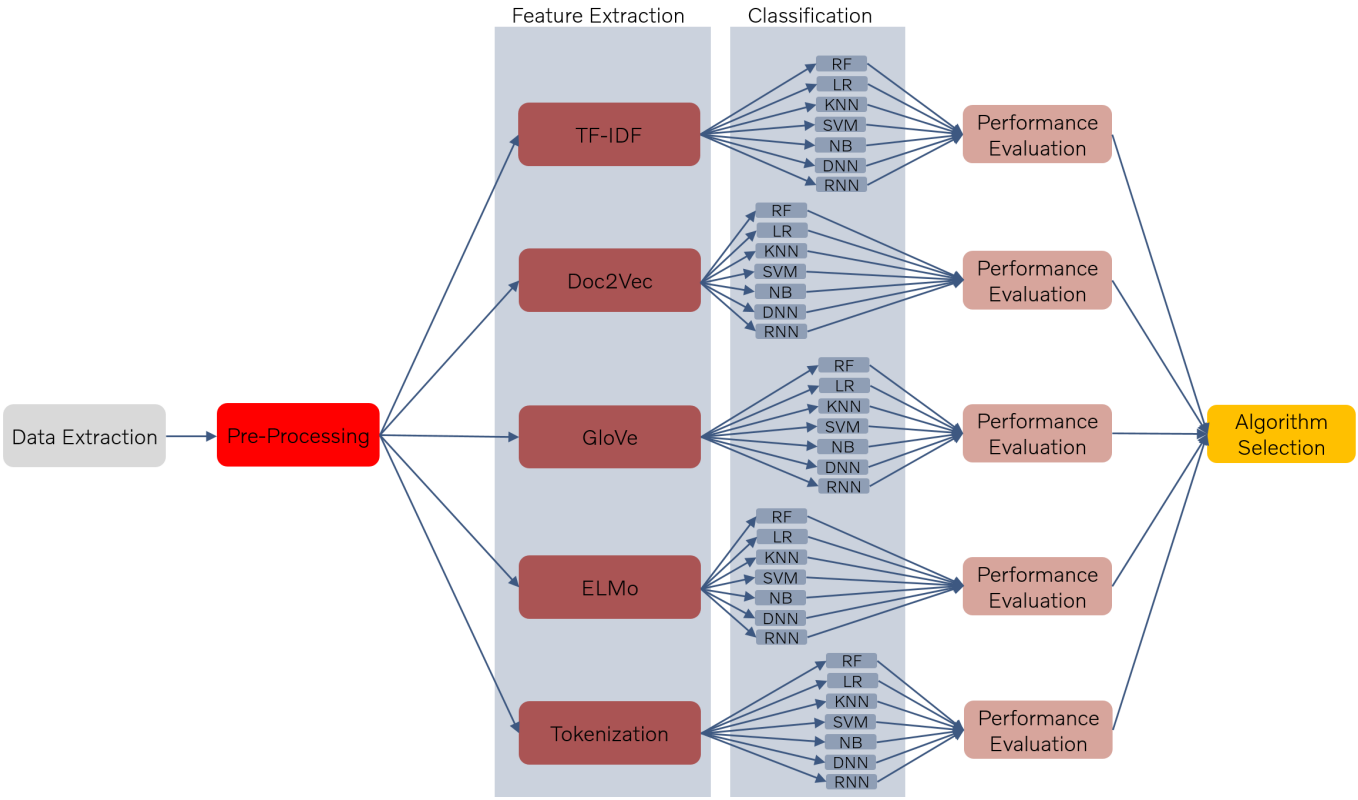


Figure 3.1: A schematic pipeline of how the different involved algorithms were combined to enable structured training and performance evaluation.

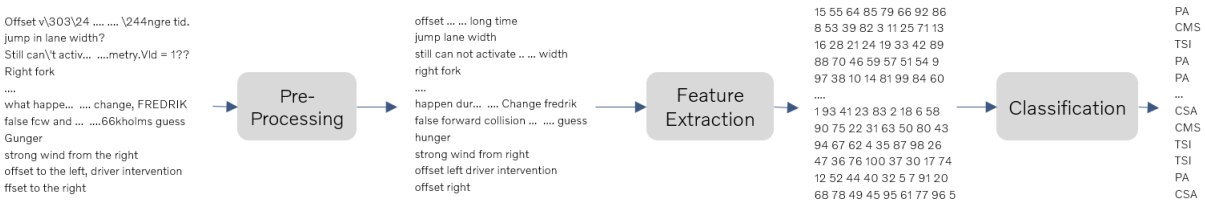


Figure 3.2: A schematic pipeline of the three main steps in the final, executable end-to-end classification algorithm solution.

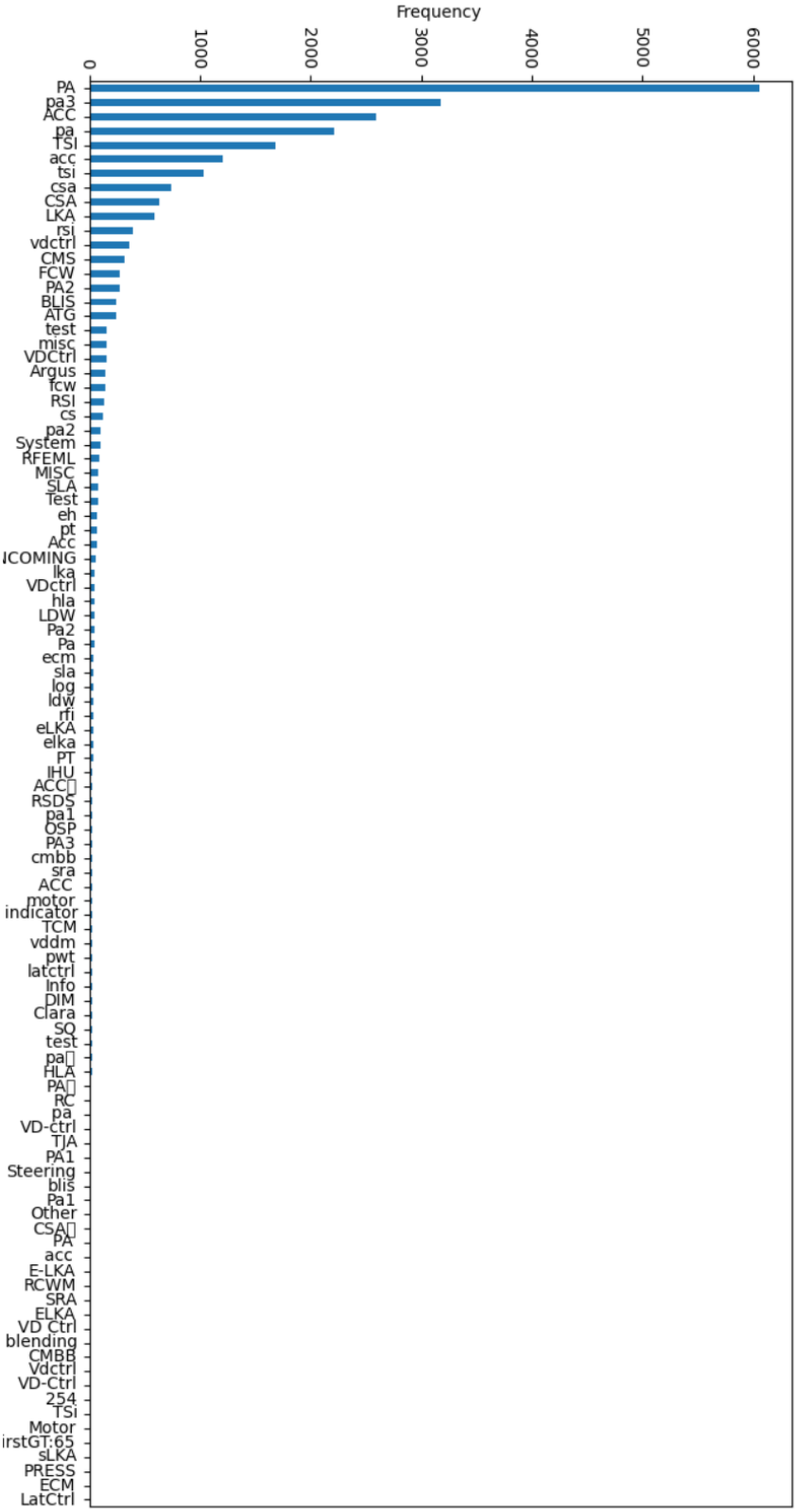


Figure 3.3: Histogram of the 100 most common classes of the auto labeled data.

the different labels were grouped together into 10 mutually exclusive main categories:

- Pilot Assist and Traffic Jam Assist (PA)
- Adaptive Cruise Control (ACC)
- Collision Mitigation Support (CMS)
- Traffic Sign Information and Road Sign Information (TSI)
- Curve Speed Assist (CSA)
- Emergency Lane Keeping Aid, Lane Keeping Aid and Blind Spot Information System (LKA)
- 12 Volt system, suspension, and miscellaneous (MISC)
- Test situations (TEST)
- Hazard Light Assist and Road Friction Estimation (CLOUD)
- Other (OTHER)

This was done in collaboration with the Volvo Car Team, so that the different auto generated labels in each class were collected together properly. A histogram of the classes, after they had been grouped, is shown in Figure 3.4.

However, since a certain amount of data in each class is required in order to train a machine learning classification algorithm properly, only those classes with at least around a thousand annotation examples were kept. These were the following:

- Pilot Assist (PA): 10523 examples
- Adaptive Cruise Control (ACC): 5383 examples
- Collision Mitigation Support (CMS): 3402 examples
- Traffic Sign Information (TSI): 3010 examples
- Curve Speed Assist (CSA): 1130 examples
- Lane Keeping Aid (LKA): 993 examples

An example of the extracted data is shown in Table 3.1.

After the data was extracted, all duplicates were removed. This reduced the available data from around 24 000 examples to 10 000 examples. The reason for the duplication filtration was that a huge amount of the annotations were automatically generated and not manually typed. Since these automatically generated annotations did not contain much useful content, a cleaner data set could be obtained by removing them. A histogram of the dataset after the duplication filtration is shown in Figure 3.5.

Finally, the data was randomly split into 80 % used for training of the algorithm, and 20 % to be used for performance evaluation of the algorithm.

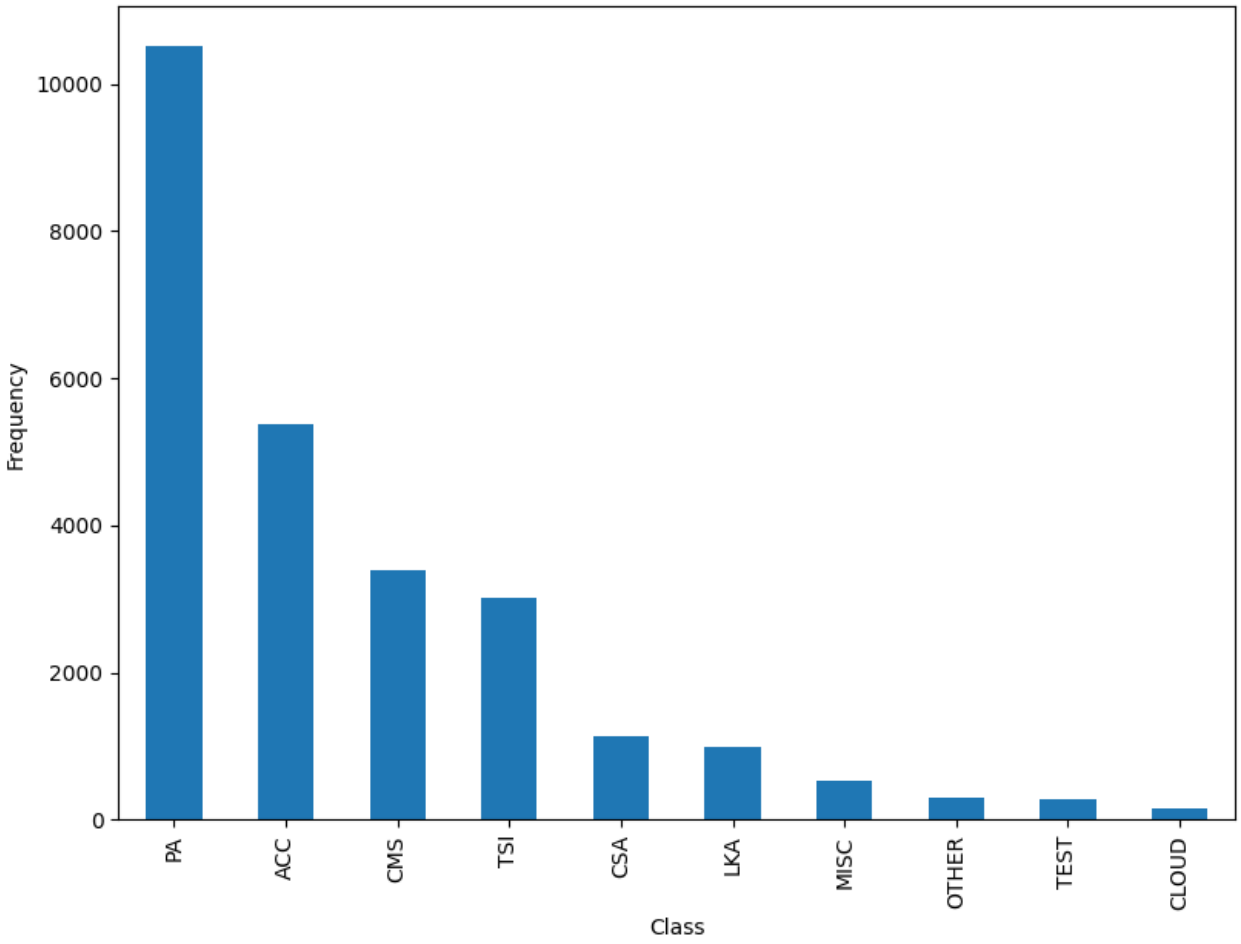


Figure 3.4: Histogram of the labeled data after the classes had been grouped together.

Table 3.1: Example of labeled annotation data

ANNOTATION	CLASS
park assist service req in toll	PA
drog på avfart	PA
styrde mot avfart	PA
hard init after lane change	PA
....	...
jerk before stop and then hard stopp	ACC
bad start. jerk	ACC
very bad stop. gear chift	ACC
tq loss at drive away	ACC
....	...
close to RE nointv	LKA
close to RE TP. vi fick meddelande och ljud o s...	LKA
testtryck	LKA
andsoff med gra ratt PA standby. ger forst rod..	LKA
....	...
Driver deactivation	CSA
Signal GrdtOfALgt not mapped correctly	CSA
Driver override accelerating lead vehicle	CSA
Target Selection: Multiple target ID changes du...	CSA
....	...
Driver override accelerating lead vehicle	TSI
Acceleration above acceleration tube for m...	TSI
Real time gap below 0.4s	TSI
Cancelled ESC intervention	TSI
....	...
Very early FCW just before	CMS
FCW intervention	CMS
Driver activation request denied. function...	CMS
FCW: Brake pedal pressed down within 2s from FCW	CMS

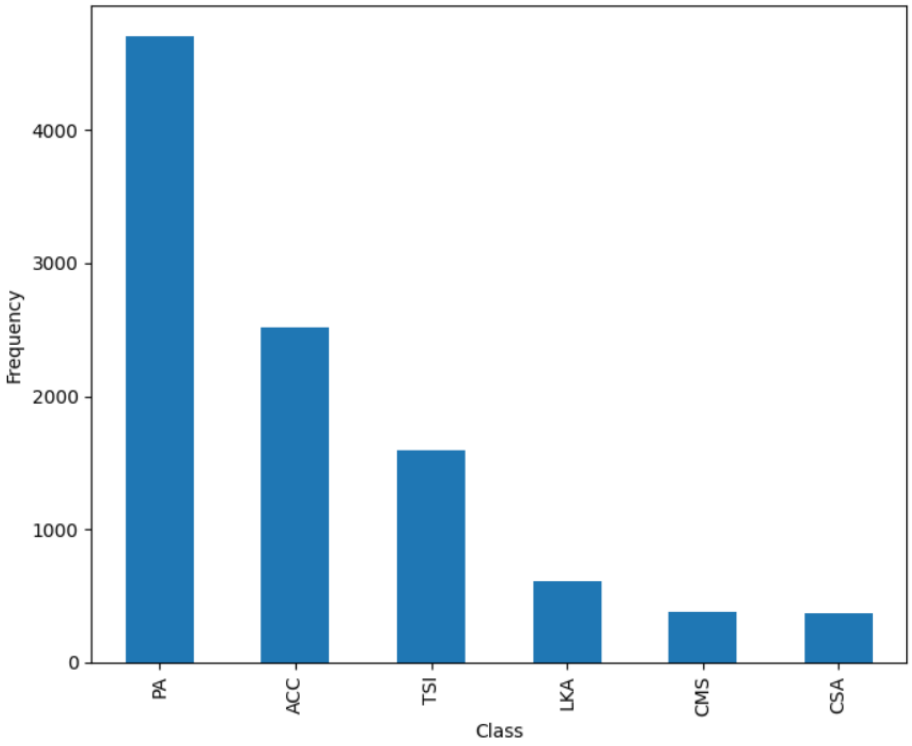


Figure 3.5: Histogram of the labeled data after duplication filtration had been executed.

3.2 Pre-Processing

Vital to all NLP problems, the text needs to be processed in order to clean up noise, correct certain errors, and translate the text to the same language. Many pre-processing algorithms exist, and a selection of what techniques to use, as well as in which order, needed to be made. This was done according to a study of Magliani, Fontanini, Fornacciari, *et al.* [43], as well with tuning and testing of what combinations that made the classification algorithm perform best. A schematic view of the pre-processing methods are represented in Figure 3.6, and each step of the processing pipeline is described in more detail below.

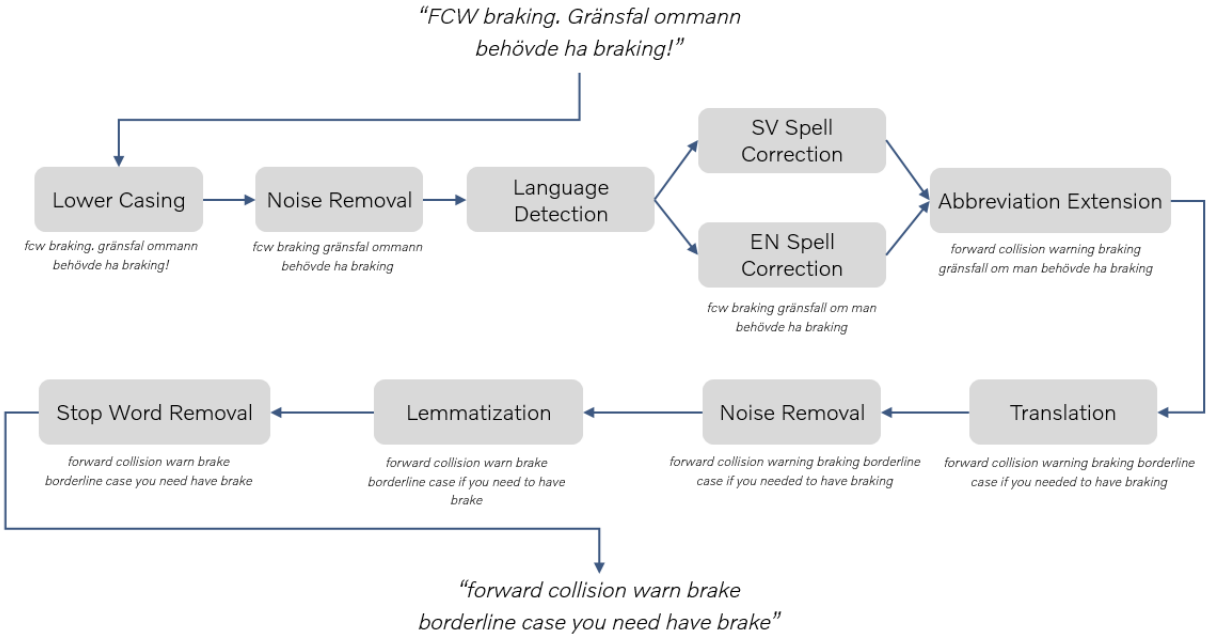


Figure 3.6: A schematic view of the pre-processing methods used.

1. Lower Casing: Converting all letter to lower case in order to reduce the number of unique characters in that data.

2. Noise Removal: Expanding contractions, for example converting "can't" into "can not" as well as removing punctuation and special characters. This was done using Python's *Regular Expression* [81] library.

3. Language Detection: Detecting the language of the text. This was done using the Python library *langdetect* [82].

4. Spelling Correction: Autocorrects the spelling of the text, depending on the previous detected language. This was done using the spellchecker included in the Python library *pyenchant* [83].

5. Abbreviation Extension: Extends certain abbreviations to its full form. The team at Volvo Cars has a lot of different abbreviations for different kinds of AD- or ADAS functionality. Some examples are “LATCTRL” (lateral control), “TSI” (traffic sign information), “EH” (electronic horizon) and “SRA” (slippery road alert). All common abbreviations were collected in a dictionary, in collaboration with the team at Volvo Cars. Every word that was a key in this dictionary was then extended to the corresponding value.

6. Translation: If the text previously was detected as anything else than English, it was translated to English using Azure Translator [84].

7. Noise Removal: Once more, a noise removal step was performed in order to reduce any occurring noise from the previous steps.

8. Lemmatization: Converts each word to its base form in order to reduce the vocabulary size. This is done by grouping together all inflected forms of a word, which was performed using the *NLTK Wordnet* [85] library.

9. Stop Word Removal: As a final step, all words which do not contain any context were removed such as “is”, “has”, -“be” etc. The stop words were defined using the *NLTK Corpus* [86] library.

3.3 Feature Extraction

The second key step in natural language processing algorithms is to convert the text into numerical features. Many powerful and well known methods exist. In order to find out which method performed best for this particular text classification problem, several feature extraction algorithms were implemented in parallel. In a later stage, when the classification algorithms had been implemented, the performance difference between these methods could be evaluated to allow for structured parameter tuning and optimization. Each feature extraction method implemented is described below.

3.3.1 Tokenizer

One of the simplest form of feature extraction is to convert each word into an integer, corresponding to the index of that word in a global dictionary. This dictionary was created with the 5000 most common words in the whole data set, using Tensorflow’s *Tokenizer* library [87]. Since the number of words in each annotation vary, the data was zero padded to get an equal length of 15 indices for each annotation. For annotations containing more than 15 words, only the first 15 words were used. To be able to execute the algorithm on unseen words, the case of “Out Of Vocabulary” (OOV) needed to be handled, which was done with an additional OOV key in the dictionary. Table 3.2 contains an example of the tokenization process of a number of annotations.

Table 3.2: Example of the feature extracted annotations by the usage of tokenization

ANNOTATION	TOKENIZED ANNOTATION
park assist service req in toll	[347 3 500 1205 0 ... 0]
drog på avfart	[1 1622 1 0 0 ... 0]
styrde mot avfart	[1 2646 1 0 0 ... 0]
engine reduced performance	[71 1 357 0 0 ... 0]
hard init after lane change	[32 2966 1 12 0 ... 0]
...	...
steps in curvce	[1 1 1955 0 0 ... 0]
jerk before stop and then hard stopp	[44 1 28 1 1 ... 0]
bad start. jerk	[34 66 44 0 0 ... 0]
very bad stop. gear chift	[1 34 28 150 0 ... 0]
tq loss at drive away	[1 479 484 41 0 ... 0]

3.3.2 Term Frequency-Inverse Document Frequency (TF-IDF)

The TF-IDF model was based on the module *TfidfModel* from the Gensim library, which is an open-source library within NLP used for handling large amount of data through applying unsupervised machine learning algorithms [88]. The *TfidfModel* module took the whole corpus, in this case a list of annotations, to create a dictionary in order to calculate the local TF vectors and the global IDF vector. Every document had its corresponding TF vector, which was a vector with the same dimensions as the dictionary, namely consisting of 3586 elements. Since the annotations were considerably shorter than the length of the dictionary, only a few elements of every TF vector were filled with a positive value while the rest consisted of zeros. The non-zero values represented the frequency of each word in the document. In contrast to the large amount of TF vectors, there was only a single IDF vector representing the whole corpus. No zero-valued elements were included in the IDF vector, since they represented the inverse of how many documents each term of the dictionary occurred in. As mentioned in Section 2.2.2.1, the two frequency vectors were multiplied to create the term-weighting system TF-IDF, which was in turn normalized by dividing each term weight by the normalization formula

$$\sqrt{\sum_{k=1}^{t} w_{i_k}^2}. \tag{3.1}$$

Table 3.3 shows some examples of the TF-IDF vectors.

3.3.3 Doc2Vec

Another Gensim model, *Doc2Vec* [89], was used to implement the feature extraction method with the same name. The neural network based model can be trained using two different approaches, namely the distributed memory method or the distributed bag of words method. Both methods were tested initially to rule out the training

Table 3.3: Example of feature vectors extracted by TF-IDF. The non-zero elements consist of values between zero and one, however each vector contains mostly zeros. The values are rounded to two decimals precision.

ANNOTATION	TF-IDF VECTORS
park assist service req in toll	[0.16 0.43 0.56 0.47 ... 0]
drog på avfart	[0 ... 0 0.65 0.76 0]
styrde mot avfart	[0 ... 0.53 0 0.54 0.65]
engine reduced performance	[0 ... 0 0.44 0.62 0.65]
hard init after lane change	[0 ... 0.32 0.52 0.45 0.46]
...	...
steps in curvce	[0 0 ... 0.70 0.70 0]
jerk before stop and then hard stopp	[0 ... 0.49 0 ... 0.53]
bad start. jerk	[0 ... 0.56 0.54 ... 0]
very bad stop. gear chift	[0 ... 0.46 0.76 ... 0]
tq loss at drive away	[0 ... 0.43 0.32 ... 0]

algorithm with the worst performance. The distributed memory method showed to be the highest performing alternative of the two, and it was used for further optimization and improvements. In the optimization step, the focus laid on three model input arguments, namely: the feature vector dimension, the number of training epochs, and the frequency limit a word had to reach in order to be included in training.

The frequency limit, which was finally set to the default value of five, assured that the model did not take very uncommon words into account when training the neural network, thus avoiding that rare, and perhaps non-significant, words affected the weights of the network. The feature vector dimension was set to different powers of two to investigate the optimal size. Starting at 4, the size of the vector was increased until the performance declined. The number of epochs were initially set to 20, but it was increased successively due to an unsatisfactory level of accuracy. Even though the accuracy of the predictions went up when the amount of epochs increased, so did the computation time. The *Doc2Vec* module went over the whole corpus during one epoch to train the network, and in the consecutive step, where the trained model was loaded, it also trained on a single new document for the same amount of epochs to optimize the output feature vector. Hence, there was a need for balance between model accuracy and computation time. 500 epochs were included in the final version of the module implementation, which seemed to yield a sufficient level of accuracy, yet it did not take an immense amount of time. An example of the feature vectors produced by *Doc2Vec* can be seen in Table 3.4.

3.3.4 Global Vectors for Word Representation (GloVe)

Another word vector technique is GloVe, which similarly to Doc2Vec converts each word to a vector space in which similar words cluster together. However, GloVe has an advantage over Doc2Vec since, in addition to the local context, it incorporates

Table 3.4: Example of feature vectors extracted by Doc2Vec.

ANNOTATION	DOC2VEC VECTORS					
park assist service req in toll	[0.19	-1.91	-1.64	-0.90	-3.12	...]
drog på avfart	[-1.48	-0.54	1.48	-1.10	1.17	...]
styrde mot avfart	[-2.17	-0.22	1.87	-1.57	0.69	...]
engine reduced performance	[-0.11	0.34	-5.08	-3.21	-3.41	...]
hard init after lane change	[5.12	-5.53	-2.57	-7.60	3.75	...]
...			...			
steps in curvce	[-0.04	-0.03	0.05	0.01	-0.01	...]
jerk before stop and then hard stopp	[0.18	-3.12	-1.38	-4.17	-2.71	...]
bad start. jerk	[1.76	-1.15	1.30	-3.76	-1.10	...]
very bad stop. gear chift	[0.93	1.56	-2.61	-4.32	0.36	...]
tq loss at drive away	[-3.87	-0.74	-1.49	-2.51	-1.51	...]

also global statistics, or word co-occurrence, to obtain the word vectors. It was therefore selected as another feature extraction method to implement.

Since the available training data in this project is limited, training a GloVe algorithm from scratch would result in a very small available corpus with potential performance issues. Therefore, a transfer learning approach was chosen by using pre-trained GloVe-vectors. Pre-trained vector sets were downloaded from a website [90] provided by the authors of [6]. The model trained on Wikipedia 2014 Corpus + Gigaword 5 corpus was selected, which consists of 6 billion tokens. These were pre-trained using AdaGrad optimization, explained in Section 2.2.3.9.3. The parameters $x_{\max}$ and α were set to 100 and 3/4, respectively. Each word in an annotation was then converted to one numerical 100-element vector representation using the pre-trained GloVe model. However, since one annotation consists of many words, a weighted average of all the word vectors in each annotation was calculated. This finally resulted in one numerical 100-sized vector for each annotation.

3.3.5 Deep Contextualized Word Representations (ELMo)

Deep Contextualized Word Representations, also called ELMo, was first introduced by Peters, Neumann, Iyyer, *et al.* [53] and is a very complex feature extraction method, however proven to be powerful. It was therefore selected as one of the feature extraction method to test for this project. However, since the available amount of training data is very small, the transfer learning approach of using a pre-trained ELMo algorithm was chosen. This was implemented using *TensorFlow Hub*, with the V3 edition of the ELMo transfer learning library [91]. Since the output of the algorithm gives a word vector representation of each word in an annotation, the mean value of the ELMo representations of all words in the annotation was computed as a final step, resulting in one 1064-sized vector for each annotation.

3.4 Classification

The final section of the NLP pipeline was the classification part. Here, the numerical features were fed to a variety of classification algorithms that attempted to distinguish the differences between the feature vectors. However, note that not all classifiers were compatible with every single feature extraction method. Models of different complexity level were implemented in order to investigate which approach would work best on documents containing few words and loose structure. The hyperparameters were optimized on the basis of the models' evaluation scores. For all classifiers, balanced class weights were implemented in order to compensate for the heavily imbalanced data set.

3.4.1 Random Forest

The random forest algorithm was implemented using the method *RandomForestClassifier* [92] from the open source machine learning library scikit-learn [93], [94]. The initial implementation was mostly based on the models' default values. However, when the accuracy showed to be quite good in comparison to the other classification methods, a hyper parameter optimizer was added to increase the performance further. Five hyper parameters were used for the optimization of the classifier, namely:

- *n_estimators*: the number of trees in the forest,
- *max_features*: the maximum size of each random subset of features to be considered when splitting a node,
- *max_depth*: the maximum depth of each tree,
- *min_samples_split*: the minimum number of data points in a node before it could be split,
- *min_samples_leaf*: the minimum number of data points that had to be passed forward to a leaf node for the split to be executed.

The optimizer, called *RandomizedSearchCV* [95], was fed a set of values for each of the five hyper parameters. By randomly creating different combinations of the hyper parameter values and applying those to the classifier, the highest performing hyper parameter settings could be found. However, these settings varied depending on the content and length of the feature vectors. Hence, the *RandomizedSearchCV* was executed for every pair of feature extraction method and random forest classifier. The final settings for each combination of methods are displayed in Table 3.5.

3.4.2 Naïve Bayes (NB)

The scikit-learn module *naive_bayes* [96] was used to implement the Naïve Bayes classifier. More specifically, the submodule *ComplementNB* [97] was applied, which was originally designed to handle imbalanced data sets and correct drastic assumptions that occurred in the standard Multinomial Naïve Bayes classifier. Naïve Bayes was compatible with only two out of the five implemented feature extraction methods, and showed to perform quite poorly, hence it was only implemented using the

Table 3.5: The final hyper parameters for the Random Forest classifier in combination with different feature extraction method combinations. The values for the max_features parameter regards the binary logarithm or the square root of the size of a feature vector.

Methods: RF +	**TF-IDF**	**Doc2Vec**	**GloVe**	**ELMo**
n_estimators	750	250	750	1000
max_features	log_2	log_2	square root	square root
max_depth	None	20	40	10
min_samples_split	15	5	20	2
min_samples_leaf	1	2	1	10

default values and was not prioritized during the optimization step of the algorithm development.

3.4.3 K-Nearest Neighbor (KNN)

The module *KNeighborsClassifier* [98] from scikit-learn was used to apply the K-Nearest Neighbor classifier to the feature vectors. The optimization of the method consisted of tuning the amount of data points that were included as the neighbors of a sample point. The number of neighbors used in each case is presented in Table 3.6.

Table 3.6: K-values for the different method combinations between KNN and the feature extractions. The KNN method was not compatible with the feature extraction method TF-IDF, hence the combination was excluded in the table.

Methods: KNN +	K-value
Doc2Vec	9
GloVe	5
Tokens	3
ELMo	10

3.4.4 Support Vector Machine (SVM)

The support vector machine method was implemented using the scikit-learn submodule *SVC* [99], which stands for C-Support Vector Classification. The regularization strength of the algorithm is the inverse of the C-value, hence a smaller value implies stronger regularization. Due to the relatively extensive computation time for the method to perform well, it was not extensively addressed during the optimization part of the development phase. However, a parameter sweep was conducted on the C-value, which yielded a final number of 100.

3.4.5 Logistic Regression (LR)

LogisticRegression [100] from the scikit-learn module *linear_model* [101] was implemented using the multinomial setting in order to handle the multi-class nature of the given data. In the initial version of the implementation, mostly default values were used to investigate the basic result differences between the method combinations. One of the exceptions was the number of iterations that the solvers were allowed to cover for the result to converge. It was set to an extensive number of 10000, to avoid the algorithm to be cut off too early.

Given the first results, the two pairs including the feature extraction algorithms TF-IDF and ELMo were improved further, respectively. The optimization mainly consisted of the addition of a regularization strength parameter, *C*. The final C-values for the two method combinations can be seen in Table 3.7

Table 3.7: The C-values for the Logistic Regression classifier, for the method combinations LR + TF-IDF and LR + ELMo, respectively.

Methods: LR +	C-value
TF-IDF	6
ELMo	100

3.4.6 Deep Neural Network (DNN)

A deep neural network with a number of fully connected layers was implemented using Tensorflow Keras Sequential model [102]. Many different hyper parameters and network structures were tested, and the ones with best performance were selected. After tuning, the final neural network was chosen as the following:

- 1 input layer with 256 neurons, using activation function ReLU
- 2 hidden layers with 32 neurons each, using the activation function ReLU
- 1 output layer with 6 neurons, representing the annotation categories. The activation function was selected to Softmax in order to enable multi class selection.

The model was then trained for 100 epochs, using 0.001 as learning rate and a batch size of 32. The Adam Optimizer was selected as optimization method. Since the data set was heavily imbalanced, penalty weights was added in proportion to the available data in each category. The neural network was finally trained individually for each feature extraction method.

3.4.7 Recurrent Neural Network (RNN)

Since RNNs are one of the most popular deep learning frameworks when working with serial data, they are commonly used for NLP problems and were therefore selected as one of the classification models in this project. Using Tensorflow Keras Sequential model [102] in Python, three layers were created which after tuning were set to the following parameters:

- One Embedding layer, with a vocab size of 5000 words and an embedding dimension of 64
- One drop-out layer with a ratio of 0.5 in order to reduce overfitting,
- One bi-directional layer with the same embedding dimension, consisting of a forward-pass and a backward-pass of LSTM cells,
- One fully connected layer, to construct a 6-dimensional output and a Softmax layer to predict the probabilities of the 6 classes.

However, in order to take advantage of the serial data handling potential of the RNN structure, the network was optimized for the only feature extraction method which outputs serial data. This is the Tokenizer. All other feature extraction methods implemented uses the "bag-of-words" approach, which does not take the order of the words into consideration. These feature extraction methods were therefore considered as irrelevant for the RNN structure to be able to handle. The model was finally trained for 100 epochs with the Adam optimizer, using a batch size of 32 and 0.01 in learning rate. Since the data set was heavily imbalanced, penalty weights were added in proportion to the available data in each category.

3.5 Evaluation

Since the available training data set was heavily imbalanced, using only accuracy would result in a misleading evaluation of the performance. Instead, the F_β-score was used with $\beta = 1$, since precision and recall were considered equally important. This F1-score was then calculated for each class individually, both for the training data and unseen test data. By comparing the training and test data performance, classification algorithms with overfitting behaviors could easily be identified. Overfitting reduction techniques were then applied such as increased regularization, reduction in model complexity or increased dropout.

All possible combinations of feature extraction and classification methods were implemented, and the models with promising results were tuned in order to maximize the test data F1-scores. Finally, the selected models were put together in a final pipeline, which was constructed to predict a class given one annotation string. Since more annotation categories than the 6 that are used in this project exist, the ability to see the prediction certainty was implemented. In this way, in a future use of the algorithm, only those predictions that the algorithm is confident of can be saved, while otherwise the annotation remains unclassified.

4

Results

The results are based on the evaluation scores, which are divided into two groups: the training performance and the test performance. An overview of the accuracies, as well as the F1-scores of the different combinations of feature extraction- and classification methods, is shown in Table 4.1. As can be seen in the left panel, the computation times are in general low, although TF-IDF and ELMo paired with SVM respectively results in considerably longer computational time than the other combinations. To obtain more in-depth knowledge about the algorithm's performance on the test data, the F1-score for each separate class is presented. Visual representations of the different test data F1-scores for each feature extraction technique are presented in Figure 4.1, 4.2, 4.3, 4.4, and 4.5, where each bin represents an annotation class and each group of bins represents a classification method. Some slots are empty due to incompatibility between the specific feature extraction method and the classification method.

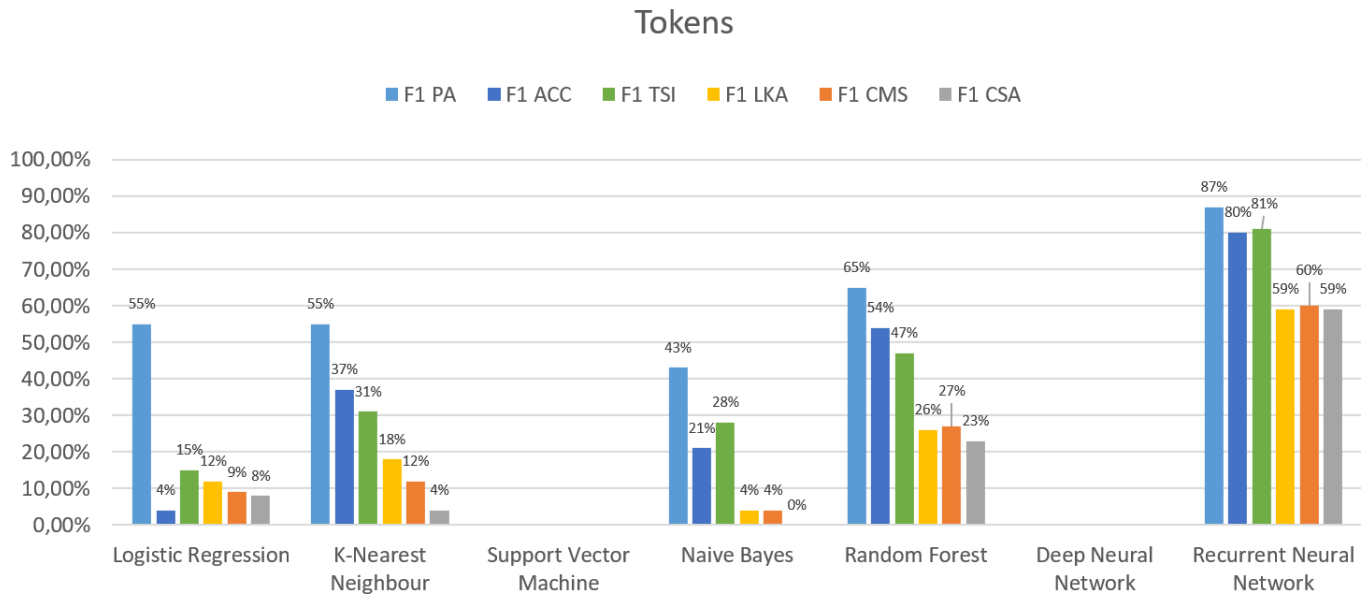


Figure 4.1: The F1-score of the *Tokenizer* feature extraction for different classification methods. Each bin represents a class.

In Figure 4.1, the combination of the feature extraction method Tokenizer and the classifier RNN clearly surpasses the performance of the other method combinations. Within each group of bins, the Pilot Assist-class seems to be the easiest to predict,

Table 4.1: An overview of the train accuracy-, test accuracy, and test F1-score performance of the text classification algorithm, using different possible combinations of feature extraction- and classification techniques. The colors represent how well the models perform, where red indicates poor results and green great performance. The upper bar includes the amount of annotations in the data set that is related to each class.

		Available Data Amount:	*4705*	*2518*	*1589*	*605*	*379*	*365*	
Model		**Time**	**Accuracy**		**Test data F1-score**				
Feat. Extr	**Classif.**	**min**	**Train**	**Test**	**PA**	**ACC**	**TSI**	**LKA**	**CMS**	**CSA**	**Average**
Tokens	LR	1	30,086%	31,136%	55,00%	4,00%	15,00%	12,00%	9,00%	8,00%	**17,17%**
	KNN	1	67,190%	41,470%	55,00%	37,00%	31,00%	18,00%	12,00%	4,00%	**26,17%**
	SVM										
	NB	1	29,955%	29,710%	43,00%	21,00%	28,00%	4,00%	4,00%	0,00%	**16,67%**
	RF	1	30,000%	49,975%	65,00%	54,00%	47,00%	26,00%	27,00%	23,00%	**40,33%**
	NN										
	RNN	10	95,423%	80,275%	87,00%	80,00%	81,00%	59,00%	60,00%	59,00%	**71,00%**
TF-IDF	LR	1	93,280%	81,900%	88,00%	80,00%	85,00%	62,00%	62,00%	60,00%	**72,83%**
	KNN										
	SVM	60	90,490%	80,370%	87,00%	79,00%	84,00%	59,00%	61,00%	57,00%	**71,17%**
	NB	3	87,980%	80,670%	88,00%	79,00%	84,00%	61,00%	61,00%	30,00%	**67,17%**
	RF	5	97,670%	83,420%	90,00%	81,00%	82,00%	66,00%	69,00%	60,00%	**74,67%**
	NN	8	98,540%	73,980%	83,00%	72,00%	78,00%	53,00%	47,00%	47,00%	**63,33%**
	RNN										
Doc2Vec	LR	5	65,687%	64,978%	74,00%	66,00%	76,00%	37,00%	41,00%	38,00%	**55,33%**
	KNN	5	76,270%	69,550%	80,00%	67,00%	69,00%	24,00%	31,00%	29,00%	**50,00%**
	SVM	5	63,820%	63,550%	76,00%	65,00%	74,00%	32,00%	31,00%	35,00%	**52,17%**
	NB										
	RF	5	97,560%	73,490%	83,00%	70,00%	80,00%	39,00%	42,00%	18,00%	**55,33%**
	NN	7	62,840%	60,110%	76,00%	61,00%	74,00%	26,00%	25,00%	26,00%	**48,00%**
	RNN										
GloVe (pre-trained)	LR	1	64,840%	62,075%	72,00%	61,00%	74,00%	35,00%	41,00%	38,00%	**53,50%**
	KNN	5	83,780%	76,290%	84,00%	71,00%	81,00%	46,00%	54,00%	56,00%	**65,33%**
	SVM	5	66,760%	63,310%	72,00%	64,00%	76,00%	34,00%	41,00%	37,00%	**54,00%**
	NB										
	RF	1	96,620%	76,680%	84,00%	71,00%	81,00%	46,00%	56,00%	53,00%	**65,17%**
	NN	5	97,030%	76,881%	84,00%	76,00%	82,00%	53,00%	55,00%	52,00%	**67,00%**
	RNN										
ELMo (pre-trained)	LR	1	95,276%	76,242%	83,00%	73,00%	85,00%	48,00%	50,00%	56,00%	**65,83%**
	KNN	15	80,500%	74,670%	84,00%	72,00%	83,00%	41,00%	48,00%	47,00%	**62,50%**
	SVM	30	97,000%	77,000%	84,00%	74,00%	84,00%	50,00%	48,00%	58,00%	**66,33%**
	NB										
	RF	1	94,680%	76,537%	83,00%	74,00%	80,00%	44,00%	50,00%	53,00%	**64,00%**
	NN	5	99,397%	81,130%	87,00%	77,00%	87,00%	57,00%	57,00%	62,00%	**71,17%**
	RNN										

however the RNN is better than any other classifier to make predictions regardless of class. Moreover, the other method combinations perform poorly in general or are not functioning at all.

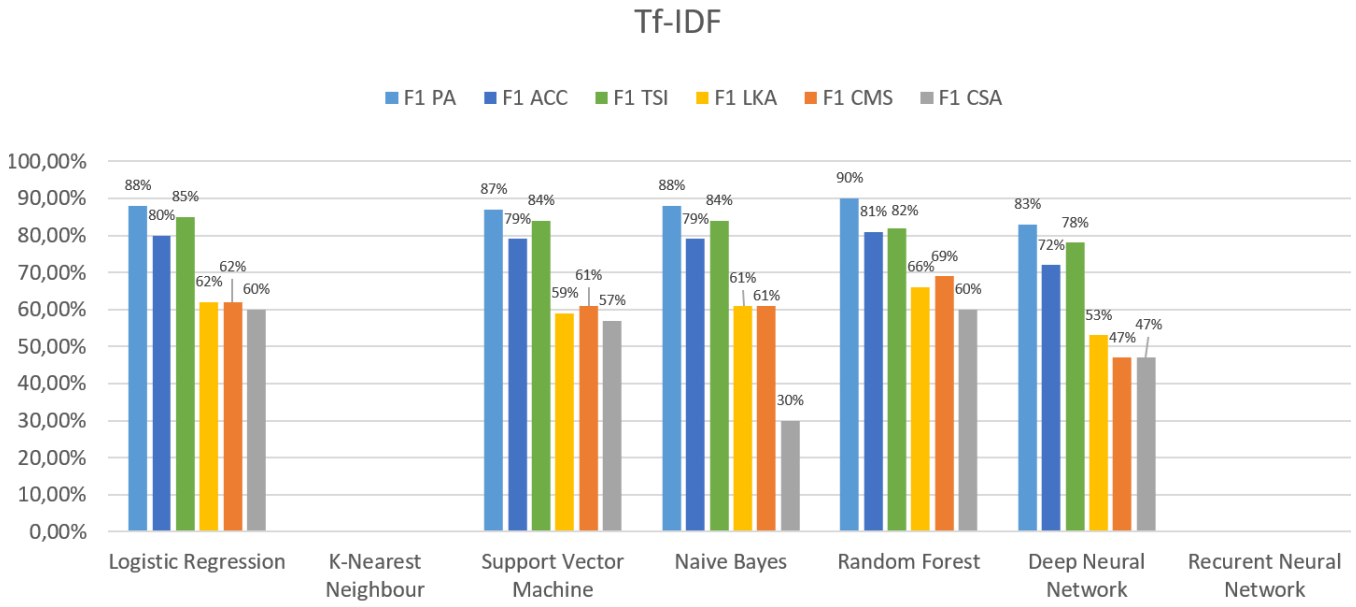


Figure 4.2: The F1-score of the *TF-IDF* feature extraction for different classification methods. Each bin represents a class.

Regarding the results in Figures 4.2, 4.3, 4.4, and 4.5, all of them display a similar prediction trend. The groups of F1-score bins all have a somewhat linearly declining appearance, where the classifiers are better at foreseeing the true class of the annotations belonging to PA , ACC and TSI compared to those that are part of LKA, CMS or CSA. The classes with the highest F1-scores are also the classes for which the data set contains the most examples, which is displayed in Table 4.1. From the same table, it can be noted that the Tokenizer performs quite poorly in combination with all classifiers except for the RNN. Moreover, the Doc2Vec feature extraction method does not perform well being combined with any of the implemented classifiers. On the contrary, TF-IDF and ELMo yield relatively good accuracies and F1-scores regardless of which classification method they are paired with. The highest performing method combination is TF-IDF + RF , which can be seen in Figure 4.2. However, the pairs TF-IDF + LR, in the same figure, Tokens + RNN in Figure 4.1, and ELMo + DNN in Figure 4.5 are not far behind.

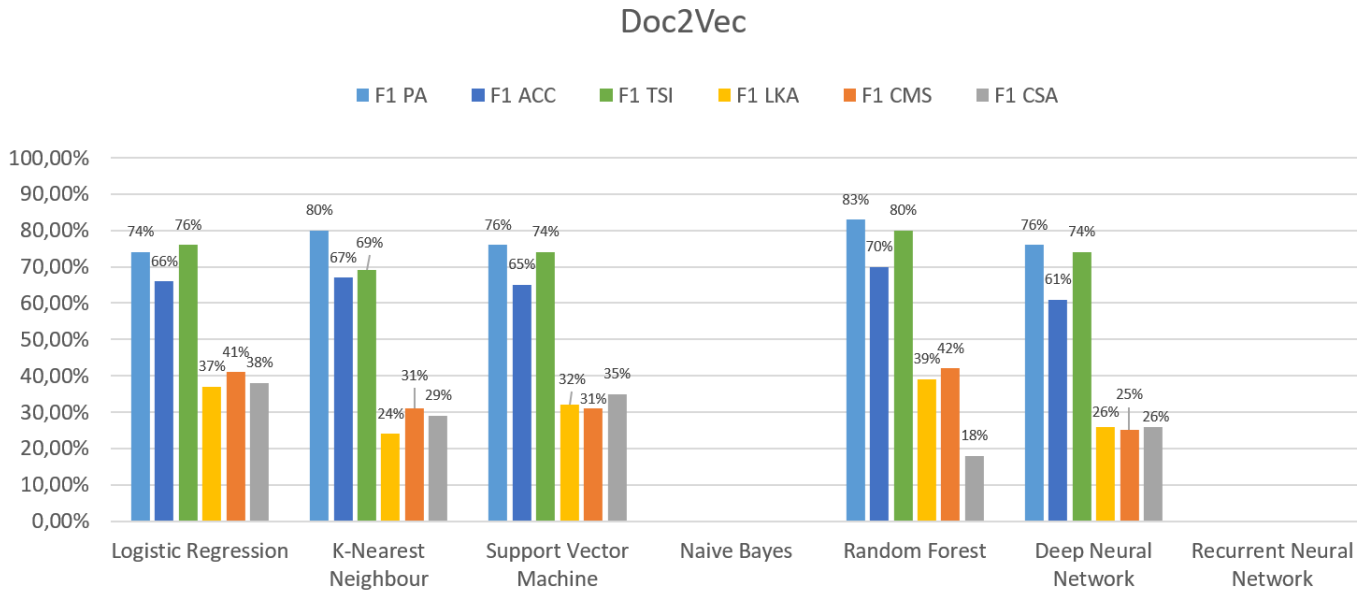


Figure 4.3: The F1-score of the *Doc2Vec* feature extraction for different classification methods. Each bin represents a class.

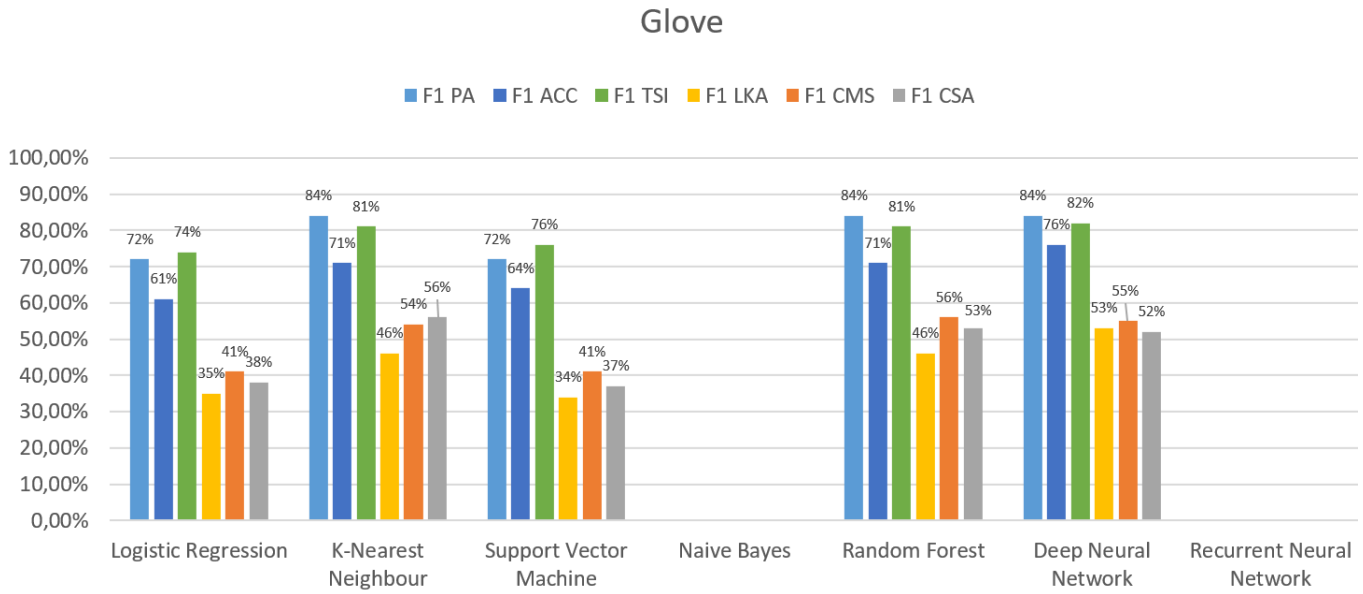


Figure 4.4: The F1-score of the *GloVe* feature extraction for different classification methods. Each bin represents a class.

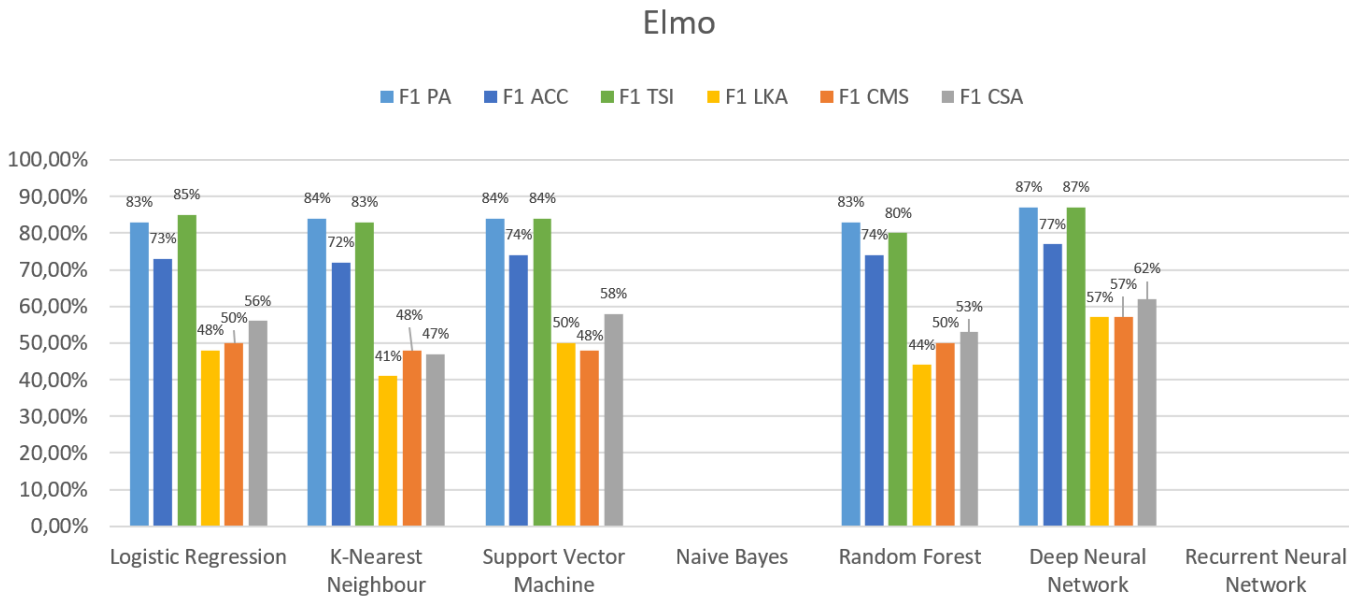


Figure 4.5: The F1-score of the *ELMo* feature extraction for different classification methods. Each bin represents a class.

5
Analysis

Looking at the results in Table 4.1, a clear correlation can be observed between the available amount of training data and the F1-score performance. Both the ACC, PA and TSI classes all have over 1500 training examples, while CMS, CSA and LKA only have a couple of hundred each. The first conclusion can therefore be made that almost all combinations of feature extraction and classification methods seem to work fine when at least 1500 training examples are available. However, the performance is reduced drastically for those classes in which the available data set is considerably smaller. The unsatisfactory part of the result can therefore be linked to the algorithm's performance on the CMS, CSA, and LKA categories. The reason why F1-scores of around 80 % are considered as "good" is due to the poor text annotation quality. Some examples are annotations like "check rejection engine", "why cancel", or "TG = 3 from now" which are hard to categorize even for AD/ADAS experienced persons at Volvo Cars. Some other annotations are also automatically generated, such as "CMSL 107. Giessen - Paris", "test" or "False", which also are completely uncorrelated to a specific class, regardless of machine learning performance. Achieving F1-scores of 80 % is therefore considered a satisfactory.

Starting out with the simplest form of feature extraction method, the Tokenizer, it can be concluded by comparing Figure 4.1 to other feature extraction methods that it performs rather poorly, regardless of the choice of classifier. This is not surprising of two reasons. First of all, Tokenization is a very simple form of feature extraction with low complexity. The ability for the majority of the implemented classification methods to infer the context of the text annotations when only an index to a dictionary is provided, is therefore expected to be low. Secondly, the Tokenization method outputs the data in series, whereas most classifiers are optimized for data based on a bag-of-words approach. However, one exception exists, which is when the Tokenizer is combined with a RNN network. It classifies the data into its different categories with surprisingly good accuracy with an average test data F1-score of 71 %, which is one of the best overall results. The reason is most probably the high complexity of the RNN structure, which is famous for its incredible capabilities of understanding natural language such as serial text data. It does however entail large complexity and require relatively long training time. Furthermore, using the Tokenization method for pre-processing is not a sustainable solution in the long-term, since it requires a corpus of the text. This is created in the training phase, and therefore only based on the available training set. More out-of-vocabulary (OOV) words, which most likely will be the case for a future, long-term use, will potentially decrease the performance of the algorithm.

The other feature extraction method that did not succeed to achieve satisfying results was Doc2Vec, also presented graphically in Figure 4.3. The best results were achieved when combined with a Random Forest classifier, but even then the F1-scores of the smaller data sets CMS, CSA and LKA, was in the unacceptable range of 20-40 %. One reason could be that Doc2Vec is a rather complex method, which trains using a neural network structure. This usually requires huge amount of training data in order to work properly, which is the most probable reason for the bad performance for this particular problem where it has been trained with low data amount. Furthermore, a possible problem regarding the generated Doc2Vec vectors is that since they are created by a neural network, the element values might not stay the same when generating vectors multiple times for the same sentence.

However, when comparing the Doc2Vec results in Figure 4.3 with the results from using GloVe in Figure 4.4, only a small performance increase can be noticed. GloVe is considered a more sophisticated method due to its incorporation of global statistics in addition to only local context, but performs in general similarly to Doc2Vec. A noticeable performance gain can however be observed in the lower data amount classes CMS, CSA and LKA. This is most probably due to the fact that a set of a pre-trained GloVe vectors on a huge Wikipedia-based data set is used, hence the feature extraction method does not lack training data. Doc2Vec, on the other hand, is trained only on the project specific data set. This is a probable reason for the noticeable performance increase on lower data amount classes when comparing Glove and Doc2Vec. It should be kept in mind however that the classification methods still are trained from scratch, hence the general trend of improved performance with increased data amount. The best result was obtained when combining GloVe with a Deep Neural Network, which surprisingly outperforms all other classification methods when using GloVe feature extraction, even for the classes with low amount of training data.

Comparing GloVe to ELMo in Figure 4.5, which is a neural-network based approach with high complexity and extensive training time, a noticeable performance improvement can be seen both in the high- and low data amount classes. Similarly to GloVe, ELMo is pre-trained on a very large data set, and the training data amount is therefore only affecting the classification methods. However, while GloVe is trained using word statistics, ELMo is based on LSTM-cells. Compared to the GloVe embeddings, which are the same static word vectors regardless of context, the ELMo embeddings can capture the context of surrounding words in a sentences. The ELMo word embedding will therefore be dynamic and adapted to the surrounding words. This inherently complex and powerful structure also results in a performance improvement, where the best result was obtained when combining ELMo with a DNN.

Out of all the implemented feature extraction methods, the TF-IDF method can preferably be compared to the structure of the Tokenizer. The values in both types of feature vectors are closely linked to the corpus, although the TF-IDF vectors contain more statistical information about the documents and words. While the Tokenizer only assigns index values to each word and form document long vectors on that basis, the TF-IDF vectors are the size of the corpus dictionary and include

values based on how frequent a word occurs within a document and how rare it is within the whole corpus. Hence, the TF-IDF contains more useful statistics, which can be the reason why it gains much higher accuracies and F1-scores across all classifiers. However, similar to the Tokenizer, TF-IDF also includes the potential drawback of the statistics being based on the current corpus. If future annotations were to include mostly new terms, that might cause more wrongly predicted classes and less accurate results, since the frequency of a specific word would be based on older data collections with different wordings.

It can also be concluded from both Table 4.1 and Figures 4.1 - 4.5 that the selection of classification method seems to affect the performance much less than the selection of feature extraction method. Some exceptions exist, however; for example the Tokenizer that barely works at all with anything else than a RNN. Moreover, both the Naive Bayes and the RNN required very specific types of input to be able to function at all, which is why their results are excluded with certain feature extraction methods. In general, however, relatively small performance improvements seemed to take place only by changing the classification method. This is also expected, since a clear data pattern, if it exists, should be discoverable by any machine learning algorithm if it is trained and tuned in a proper way. The different classification methods seem instead to differ regarding training time, execution time, and the type of input data they are able to handle best.

5.1 Future work

For future work, a data labeling process is suggested in order to gain more training data examples from all of the AD- and ADAS categories. Currently, the algorithm is trained only on six of the categories present. When the algorithm is executed to classify a new text annotation, it therefore assumes that the annotation belongs to one of these 6 classes, which is not always the case. This makes its current usage limited to current data due to that new data could contain unknown classes for the algorithm. Another reason to increase the number of training examples of all classes is to evaluate the hypothesis of a strong correlation between available training data and the classification performance, with the potential to achieve an equal F1-score for every class.

6
Conclusions

The main goal of this thesis was to create a natural language classification algorithm for categorizing short, manually written text annotations between a number of AD and ADAS related features. This was done by training and evaluating the combination of many different techniques of pre-processing, feature extraction, and classification of the text data. The best overall classification performance was achieved by implementing all of the mentioned pre-processing techniques with one of the following combinations of feature extraction and classification techniques:

- Tokenizer + Recurrent Neural Network,
- TF-IDF + Random Forest, and
- ELMo + Deep Neural Network,

all with an average test data F1-score of over 70 %. The ELMo + DNN combination is however considered as the most long-term sustainable one, since it does not depend on a corpus generated solely from the training data. Hence, it does not suffer from the risk of an increasing amount of out-of-vocabulary words in the future. It is however one of the most computationally expensive algorithms implemented, but this is not considered as a problem since the execution time when classifying new data is still in the range of a couple of seconds per annotation.

The current algorithm works well for the higher data amount classes PA, ACC, and TSI, but suffers from poor performance for the lower data amount classes LKA, CMS, and CSA. To be able to overcome this problem, the first step suggested for future work is to train the algorithm with more data examples from these particular classes.

Bibliography

[1] C. C. Aggarwal and C. Zhai, "A survey of text classification algorithms," in *Mining text data*, Springer, 2012, pp. 163–222.

[2] K. Kowsari, K. Jafari Meimandi, M. Heidarysafa, S. Mendu, L. Barnes, and D. Brown, "Text classification algorithms: A survey," *Information*, vol. 10, no. 4, p. 150, 2019.

[3] M. M. Mirończuk and J. Protasiewicz, "A recent overview of the state-of-the-art elements of text classification," *Expert Systems with Applications*, vol. 106, pp. 36–54, 2018.

[4] G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval," *Information processing & management*, vol. 24, no. 5, pp. 513–523, 1988.

[5] Y. Goldberg and O. Levy, "Word2vec explained: Deriving mikolov et al.'s negative-sampling word-embedding method," *arXiv preprint arXiv:1402.3722*, 2014.

[6] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.

[7] H. Abdi and L. J. Williams, "Principal component analysis," *Wiley interdisciplinary reviews: computational statistics*, vol. 2, no. 4, pp. 433–459, 2010.

[8] M. Sugiyama, "Dimensionality reduction of multimodal labeled data by local fisher discriminant analysis.," *Journal of machine learning research*, vol. 8, no. 5, 2007.

[9] S. Tsuge, M. Shishibori, S. Kuroiwa, and K. Kita, "Dimensionality reduction using non-negative matrix factorization for information retrieval," in *2001 IEEE International Conference on Systems, Man and Cybernetics. e-Systems and e-Man for Cybernetics in Cyberspace (Cat. No. 01CH37236)*, IEEE, vol. 2, 2001, pp. 960–965.

[10] E. Bingham and H. Mannila, "Random projection in dimensionality reduction: Applications to image and text data," in *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, 2001, pp. 245–250.

[11] W. Wang, Y. Huang, Y. Wang, and L. Wang, "Generalized autoencoder: A neural network framework for dimensionality reduction," in *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, 2014, pp. 490–497.

[12] L. Van der Maaten and G. Hinton, "Visualizing data using t-sne.," *Journal of machine learning research*, vol. 9, no. 11, 2008.

[13] J. Rocchio, "Relevance feedback in information retrieval," *The Smart retrieval system-experiments in automatic document processing*, pp. 313–323, 1971.

[14] R. Farzi and V. Bolandi, "Estimation of organic facies using ensemble methods in comparison with conventional intelligent approaches: A case study of the south pars gas field, persian gulf, iran," *Modeling Earth Systems and Environment*, vol. 2, no. 2, pp. 1–13, 2016.

[15] D. R. Cox and E. J. Snell, *Analysis of binary data*. CRC press, 1989, vol. 32.

[16] Z. Qu, X. Song, S. Zheng, X. Wang, X. Song, and Z. Li, "Improved bayes method based on tf-idf feature and grade factor feature for chinese information classification," in *2018 IEEE International Conference on Big Data and Smart Computing (BigComp)*, IEEE, 2018, pp. 677–680.

[17] S. Jiang, G. Pang, M. Wu, and L. Kuang, "An improved k-nearest-neighbor algorithm for text categorization," *Expert Systems with Applications*, vol. 39, no. 1, pp. 1503–1509, 2012.

[18] B. E. Boser, I. M. Guyon, and V. N. Vapnik, "A training algorithm for optimal margin classifiers," in *Proceedings of the fifth annual workshop on Computational learning theory*, 1992, pp. 144–152.

[19] J. N. Morgan and J. A. Sonquist, "Problems in the analysis of survey data, and a proposal," *Journal of the American statistical association*, vol. 58, no. 302, pp. 415–434, 1963.

[20] T. K. Ho, "Random decision forests," in *Proceedings of 3rd international conference on document analysis and recognition*, IEEE, vol. 1, 1995, pp. 278–282.

[21] K. Kowsari, D. E. Brown, M. Heidarysafa, K. J. Meimandi, M. S. Gerber, and L. E. Barnes, "Hdltex: Hierarchical deep learning for text classification," in *2017 16th IEEE international conference on machine learning and applications (ICMLA)*, IEEE, 2017, pp. 364–371.

[22] J. Lever, M. Krzywinski, and N. Altman, *Classification evaluation*, 2016.

[23] B. W. Matthews, "Comparison of the predicted and observed secondary structure of t4 phage lysozyme," *Biochimica et Biophysica Acta (BBA)-Protein Structure*, vol. 405, no. 2, pp. 442–451, 1975.

[24] A. P. Yonelinas and C. M. Parks, "Receiver operating characteristics (rocs) in recognition memory: A review.," *Psychological bulletin*, vol. 133, no. 5, p. 800, 2007.

[25] J. Zeng, J. Li, Y. Song, C. Gao, M. R. Lyu, and I. King, "Topic memory networks for short text classification," *arXiv preprint arXiv:1809.03664*, 2018.

[26] J. Wang, Z. Wang, D. Zhang, and J. Yan, "Combining knowledge with deep convolutional neural networks for short text classification.," in *IJCAI*, vol. 350, 2017.

[27] J. Howard and S. Ruder, "Universal language model fine-tuning for text classification," *arXiv preprint arXiv:1801.06146*, 2018.

[28] A. Kaplan and M. Haenlein, "Siri, siri, in my hand: Who's the fairest in the land? on the interpretations, illustrations, and implications of artificial intelligence," *Business Horizons*, vol. 62, no. 1, pp. 15–25, 2019.

[29] T. Mitchell, "Machine learning, mcgraw-hill higher education," *New York*, 1997.

[30] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org.

[31] M. Kubat, *An introduction to machine learning*. Springer, 2017.

[32] X. Zhu, "Semi-supervised learning," in *Encyclopedia of Machine Learning and Data Mining*, C. Sammut and G. Webb, Eds., 2017.

[33] L. Marquez and J. G. Salgado, "Machine learning and natural language processing," 2000.

[34] J. Read, B. Pfahringer, G. Holmes, and E. Frank, "Classifier chains for multi-label classification," *Machine learning*, vol. 85, no. 3, p. 333, 2011.

[35] R. Sergienko, M. Shan, and A. Schmitt, "A comparative study of text preprocessing techniques for natural language call routing," in *Dialogues with Social Robots*, ser. Lecture Notes in Electrical Engineering, K. Jokinen and G. Wilcock, Eds., vol. 427, 2016. DOI: https://doi.org/10.1007/978-981-10-2585-3_2.

[36] B. Pahwa, S Taruna, and N. Kasliwal, "Sentiment analysis-strategy for text pre-processing," *Int. J. Comput. Appl*, vol. 180, pp. 15–18, 2018.

[37] S. Dhuliawala, D. Kanojia, and P. Bhattacharyya, "Slangnet: A wordnet like resource for english slang," in *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*, 2016, pp. 4329–4332.

[38] M. K. Dalal and M. A. Zaveri, "Automatic text classification: A technical review," *International Journal of Computer Applications*, vol. 28, no. 2, pp. 37–40, 2011.

[39] H. Saif, M. Fernandez, Y. He, and H. Alani, "On stopwords, filtering and data sparsity for sentiment analysis of twitter," 2014.

[40] V. M. Christanti, D. S. Naga, *et al.*, "Fast and accurate spelling correction using trie and damerau-levenshtein distance bigram," *Telkomnika*, vol. 16, no. 2, pp. 827–833, 2018.

[41] F. J. Damerau, "A technique for computer detection and correction of spelling errors," *Communications of the ACM*, vol. 7, no. 3, pp. 171–176, 1964.

[42] K. Spirovski, E. Stevanoska, A. Kulakov, Z. Popeska, and G. Velinov, "Comparison of different model's performances in task of document classification," in *Proceedings of the 8th International Conference on Web Intelligence, Mining and Semantics*, 2018, pp. 1–12.

[43] F. Magliani, T. Fontanini, P. Fornacciari, S. Manicardi, and E. Iotti, "A comparison between preprocessing techniques for sentiment analysis in twitter," Dec. 2016.

[44] S. Pradha, M. N. Halgamuge, and N. Tran Quoc Vinh, "Effective text data preprocessing technique for sentiment analysis in social media data," in *2019 11th International Conference on Knowledge and Systems Engineering (KSE)*, 2019, pp. 1–8. DOI: 10.1109/KSE.2019.8919368.

[45] T. Bolukbasi, K.-W. Chang, J. Zou, V. Saligrama, and A. Kalai, "Man is to computer programmer as woman is to homemaker? debiasing word embeddings," *arXiv preprint arXiv:1607.06520*, 2016.

[46] I. A. El-Khair, "Term weighting.," *Encyclopedia of database systems*, vol. 1, pp. 3037–3040, 2009.

[47] K. S. Jones, "A statistical interpretation of term specificity and its application in retrieval," *Journal of documentation*, 1972.

[48] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[49] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *arXiv preprint arXiv:1310.4546*, 2013.

[50] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *International conference on machine learning*, PMLR, 2014, pp. 1188–1196.

[51] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American society for information science*, vol. 41, no. 6, pp. 391–407, 1990.

[52] E. F. Sang and F. De Meulder, "Introduction to the conll-2003 shared task: Language-independent named entity recognition," *arXiv preprint cs/0306050*, 2003.

[53] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep contextualized word representations," *arXiv preprint arXiv:1802.05365*, 2018.

[54] O. Melamud, J. Goldberger, and I. Dagan, "Context2vec: Learning generic context embedding with bidirectional lstm," in *Proceedings of the 20th SIGNLL conference on computational natural language learning*, 2016, pp. 51–61.

[55] M. E. Peters, W. Ammar, C. Bhagavatula, and R. Power, "Semi-supervised sequence tagging with bidirectional language models," *arXiv preprint arXiv:1705.00108*, 2017.

[56] B. McCann, J. Bradbury, C. Xiong, and R. Socher, "Learned in translation: Contextualized word vectors," *arXiv preprint arXiv:1708.00107*, 2017.

[57] M Ikonomakis, S. Kotsiantis, and V Tampakas, "Text classification using machine learning techniques.," *WSEAS transactions on computers*, vol. 4, no. 8, pp. 966–974, 2005.

[58] L. Breiman, "Bagging predictors," *Machine learning*, vol. 24, no. 2, pp. 123–140, 1996.

[59] R. E. Schapire, "A brief introduction to boosting," in *Ijcai*, Citeseer, vol. 99, 1999, pp. 1401–1406.

[60] E. Bauer and R. Kohavi, "An empirical comparison of voting classification algorithms: Bagging, boosting, and variants," *Machine learning*, vol. 36, no. 1, pp. 105–139, 1999.

[61] S. R. Safavian and D. Landgrebe, "A survey of decision tree classifier methodology," *IEEE transactions on systems, man, and cybernetics*, vol. 21, no. 3, pp. 660–674, 1991.

[62] R. L. De Mántaras, "A distance-based attribute selection measure for decision tree induction," *Machine learning*, vol. 6, no. 1, pp. 81–92, 1991.

[63] S. Kaufmann, "Cuba: Artificial conviviality and user-behaviour analysis in web-feeds," Ph.D. dissertation, University of Luxembourg, Luxembourg, Luxembourg, 2010.

[64] E. S. Pearson, "Bayes' theorem, examined in the light of experimental sampling," *Biometrika*, pp. 388–442, 1925.

[65] W. Zhang and F. Gao, "An improvement to naive bayes for text classification," *Procedia Engineering*, vol. 15, pp. 2160–2164, 2011.

[66] N Suguna and K Thanushkodi, "An improved k-nearest neighbor classification using genetic algorithm," *International Journal of Computer Science Issues*, vol. 7, no. 2, pp. 18–21, 2010.

[67] S. Tong and D. Koller, "Support vector machine active learning with applications to text classification," *Journal of machine learning research*, vol. 2, no. Nov, pp. 45–66, 2001.

[68] C. J. Burges, "A tutorial on support vector machines for pattern recognition," *Data mining and knowledge discovery*, vol. 2, no. 2, pp. 121–167, 1998.

[69] A. Juan and E. Vidal, "On the use of bernoulli mixture models for text classification," *Pattern Recognition*, vol. 35, no. 12, pp. 2705–2710, 2002.

[70] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[71] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv preprint arXiv:1412.3555*, 2014.

[72] S. Albawi, T. A. Mohammed, and S. Al-Zawi, "Understanding of a convolutional neural network," in *2017 International Conference on Engineering and Technology (ICET)*, Ieee, 2017, pp. 1–6.

[73] S. Lai, L. Xu, K. Liu, and J. Zhao, "Recurrent convolutional neural networks for text classification," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 29, 2015.

[74] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010*, Springer, 2010, pp. 177–186.

[75] T. Tieleman and G Hinton, "Divide the gradient by a running average of its recent magnitude. coursera neural netw," *Mach. Learn*, vol. 6, pp. 26–31, 2012.

[76] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization.," *Journal of machine learning research*, vol. 12, no. 7, 2011.

[77] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[78] M. D. Zeiler, "Adadelta: An adaptive learning rate method," *arXiv preprint arXiv:1212.5701*, 2012.

[79] Y. Yang, "An evaluation of statistical approaches to text categorization," *Information retrieval*, vol. 1, no. 1, pp. 69–90, 1999.

[80] *Python*, Accessed: 2021-04-28. [Online]. Available: `https://www.python.org/`.

[81] *Regular expression*, Accessed: 2021-05-13. [Online]. Available: `https://docs.python.org/3/library/re.html`.

[82] *Langdetect*, Accessed: 2021-04-28. [Online]. Available: https://pypi.org/project/langdetect/.
[83] *Pyenchant*, Accessed: 2021-04-28. [Online]. Available: https://pyenchant.github.io/pyenchant/index.html.
[84] *Azure translator*. [Online]. Available: https://azure.microsoft.com/en-us/services/cognitive-services/translator/.
[85] *Nltk wordnet*. [Online]. Available: https://www.nltk.org/_modules/nltk/stem/wordnet.html.
[86] *Nltk corpus*. [Online]. Available: http://www.nltk.org/api/nltk.corpus.
[87] *Tensorflow tokenizer*, Accessed: 2021-04-29. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/text/Tokenizer.
[88] *What is gensim?* Accessed: 2021-04-28. [Online]. Available: https://radimrehurek.com/gensim/intro.html#what-is-gensim.
[89] *Doc2vec paragraph embeddings*, Accessed: 2021-05-03. [Online]. Available: https://radimrehurek.com/gensim/models/doc2vec.html.
[90] *Glove*, Accessed: 2021-05-04. [Online]. Available: https://nlp.stanford.edu/projects/glove/.
[91] *Elmo v3 - tensorflow hub*. [Online]. Available: https://tfhub.dev/google/elmo/3.
[92] *Sklearn.ensemble.randomforestclassifier*, Accessed: 2021-05-12. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html.
[93] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.
[94] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux, "API design for machine learning software: Experiences from the scikit-learn project," in *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, 2013, pp. 108–122.
[95] *Sklearn.model_selection.randomizedsearchcv*, Accessed: 2021-05-12. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html#sklearn.model_selection.RandomizedSearchCV.
[96] *Sklearn.naive_bayes*, Accessed: 2021-05-12. [Online]. Available: https://scikit-learn.org/stable/modules/classes.html#module-sklearn.naive_bayes.
[97] *Sklearn.naive_bayes.complementnb*, Accessed: 2021-05-12. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.ComplementNB.html#sklearn.naive_bayes.ComplementNB.
[98] *Sklearn.neighbors.kneighborsclassifier*, Accessed: 2021-05-12. [Online]. Available: https://scikit-learn.org/stable/modules/generated/

sklearn.neighbors.KNeighborsClassifier.html#sklearn.neighbors.KNeighborsClassifier.

[99] *Sklearn.svm.svc*, Accessed: 2021-05-12. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC.

[100] *Sklearn.linear_model.logisticregression*, Accessed: 2021-05-12. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html#sklearn.linear_model.LogisticRegression.

[101] *Sklearn.linear_model*, Accessed: 2021-05-12. [Online]. Available: https://scikit-learn.org/stable/modules/classes.html#module-sklearn.linear_model.

[102] *Keras sequential model*, Accessed: 2021-05-05. [Online]. Available: https://www.tensorflow.org/guide/keras/sequential_model.

DEPARTMENT OF ELECTRICAL ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY