university of groningen

faculty of science and engineering

MASTER'S THESIS

Efficient Classification and Unsupervised Keyphrase Extraction for Web Pages

Tim Haarman
s2404184

Department of Artificial Intelligence
University of Groningen, The Netherlands

Primary supervisor:
Dr. M.A. Wiering
(University of Groningen)

External Supervisor:
Drs. B.M. Zijlema
(Dataprovider.com)

July 10, 2019

Abstract

With the ever increasing size of the World Wide Web, indexing and searching through all Web pages is becoming increasingly difficult. An effective analysis and categorization of Web content therefore requires an automated approach. However, the open nature of the Web causes it to be plagued by noise and general inconsistency, making it difficult for heuristic approaches to be effective. In this thesis we investigate how information can effectively be extracted from Web pages in two tasks: *keyphrase extraction* and *Web page classification*.

Keyphrase extraction is a popular field in natural language processing (NLP), although it is rarely applied to Web pages. We therefore propose a novel unsupervised keyphrase extraction method called WebEmbedRank, which is specialized at extracting keyphrases from hypertext documents. This method combines the extraction power of a state-of-the-art keyphrase extraction method with a weighting process based on the structural information in the HTML code of a Web page. To evaluate this new method we created a gold-standard dataset, and used this to compare the results of WebEmbedRank with several state-of-the-art keyphrase extraction methods. This showed that our newly proposed method was significantly better at extracting keyphrases from Web pages compared to all other methods.

We furthermore compared several methods to automatically categorize Web pages. Multiple baseline models, convolutional and recurrent neural networks were trained on a dataset consisting of more than one million company Web pages with corresponding industrial category labels. This showed that overall the recurrent neural networks achieved the highest classification scores. Especially the model with GRU cells was found to be a good choice, as it achieved scores similar to the LSTM but required less time to make new predictions. While the recurrent neural networks generally achieved the highest classification scores, it came at the cost of highly increased prediction times for classifying new samples compared to the baselines and simpler feed forward neural networks. Finally, we also tested the impact of different pre-trained word embeddings on the classification. We found that the choice of word embedding model did not seem to have a large effect on the results, indicating that they generalize well for the current task.

Acknowledgements

First and foremost I want to thank my primary supervisor Dr. Marco Wiering, whom I could always reach out to for advice. I also want to express my gratitude to my external supervisor Bastiaan Zijlema, and to all my other coworkers at Dataprovider.com. They gave me new insights for my project, helped with constructing the keyphrase evaluation dataset, and generally made my time working on this thesis genuinely enjoyable.

Furthermore, I want to thank my parents for their unrelenting support, without whom I never would have gotten to this point. Last, but most certainly not least, I want to thank my girlfriend Maureen. She offered tremendous help in proofreading the early versions of this thesis, and provided me with an invaluable source of motivation throughout the project.

Tim Haarman
July 10, 2019

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The World Wide Web is one of the largest sources of publicly available textual data, with estimates of the size of the indexed Web ranging up to 50 billion pages [8]. Despite its size, sets of Web pages that span multiple domains are not frequently used in natural language processing (NLP) tasks. A possible explanation for this can be found in the lack of a consistent structure in Web pages. Corpora of old books, tweets or articles from static domains such as Wikipedia generally provide consistently structured, continuous and relatively clean sources of data, which are therefore easier to analyze and use than a set of Web pages that highly vary in length, structure, and content. As there are many of these corpora available, noisy and inconsistent sets of data such as a corpus of cross-domain Web pages are often avoided.

Nevertheless, being able to accurately analyze and categorize Web pages can be of great importance for many services. For example, being able to automatically produce fitting keyphrases and category labels for Web pages can make the World Wide Web easily searchable. This also makes it an interesting topic for Dataprovider.com, which is a company that indexes and structures the data on the Web to make it more accessible and insightful. They offer a large variety of information about the content of the World Wide Web, such as where Web pages are hosted, whether they are an e-commerce store, if they have SSL certification, and much more. At the time of writing Dataprovider.com has indexed and analyzed more than 281 million websites. This makes finding a method that can automatically and accurately categorize the Web an interesting task for Dataprovider.com, as it helps with further structuring their data.

In essence, this problem can be treated as any other NLP task, viewing the Web page as a text document. Two fields in NLP that are relevant for this are keyphrase extraction and document classification. Each of these are active fields of research, which have seen many recent developments. A good example of this is the introduction of word embeddings. Word embedding techniques map individual words to dense, continuous vectors in which the semantic relations between words are represented. Although the original concept has existed for longer, word embedding techniques started to become popular in 2013 with the release of Word2vec [62]. Since then, they have been used in some form in a large variety of subfields in NLP research [52].

This new way of encoding text has also sparked new innovation in the approaches to unsupervised keyphrase extraction. In this field graph-based methods have been popular for a long time [59, 53, 11, 10]. These methods often utilize word co-occurrence information from the document to rank keyphrases based on their relevance. Recently a new method was proposed that improved on the state-of-the-art graph-based methods using word embeddings [5]. Word embeddings encode the semantic relations between words, which can be used to select fitting keyphrases. By comparing each candidate keyphrase from a text with their similarity to the document as a whole, it is possible to extract only the most relevant phrases. This new method led to state-of-the-art results on several keyphrase extraction tasks.

Being able to encode the meaning of words has also caused document classification methods to largely shift towards using word embeddings, which has since become the standard way to encode text documents. The low dimensional and dense representation makes many classifiers easier to train, and the encoded relations have shown to improve classification results [47]. The ability to create semantic embeddings on the word or even character level has furthermore enabled more complex approaches, such as the use of convolutional and recurrent neural networks.

However, the vast majority of new NLP research is focused on standard text documents like journal articles or news stories. As Web pages are made up of hypertext, their contents are inherently different from most other types of documents. Apart from the aforementioned inconsistency, Web pages often contain many small snippets of texts, and may contain large amounts of irrelevant text. It is therefore not directly evident that the results from the current state-of-the-art methods transfer well to Web content. We therefore want to investigate how we can parse and extract useful information from Web pages, and how well existing methods transfer to this domain. The main question this thesis aims to answer is formulated as follows:

How can Web pages be indexed effectively and efficiently based on the available textual and structural information?

In order to find an answer to this, we break down the problem into the following secondary questions:

1. How can the most relevant keyphrases be extracted from a Web page?
2. Which classification methods can most accurately predict Web page categories?
3. What is the effect of different pre-trained word embedding models on classification?
4. Are the state-of-the-art classification methods computationally efficient enough to be used in practice in large scale applications?

This thesis attempts to answer these questions in two main parts, being *keyphrase extraction* and *Web page classification*.

In the first part we look at the best way to extract keyphrases from Web pages. The availability of relevant keyphrases can enable effective searching for specific Web pages across a large corpus. Additionally, keyphrases can provide a quick summary of any given Web page. This process of assigning the most important keyphrases to documents is a task humans are good at, but is also a highly laborious process. With collections of Web pages growing to the size of hundred millions of pages manual annotation becomes unfeasible, requiring an automated approach. In the past various methods for extracting keyphrases from documents have been proposed, but very few can make use of the extra structure an HTML page provides. We therefore propose WebEmbedRank, a novel approach that incorporates the positional information extracted from the HTML into a state-of-the-art keyphrase extraction method. We furthermore show that this method significantly increases the accuracy of the extracted keyphrases compared to several baselines and other state-of-the-art methods that do not incorporate structural information.

In the second part several approaches to classify Web pages are investigated. Over the past years a myriad of document classification methods have been proposed, however they are rarely applied to unconstrained domains such as Web pages. These methods are usually evaluated based on clean datasets. Contrary to this, the text that can be retrieved from Web pages can be highly noisy, for example due to broken HTML code or large sections of text that are not directly relevant to the type of Web page. Additionally, the individual text segments are often very short and inconsistently written across Web domains. Due to the size of the Web and its high variability it is not possible to either manually check all Web pages, or to devise some preprocessing steps that can filter out all noise. In this thesis we therefore aim to find how well these methods perform outside of a controlled experimental setting, where some noise in the input is unavoidable. Additionally, we are also interested in the computational efficiency of the methods when making predictions for new samples. The focus in the past years has largely shifted towards very deep neural networks, and while these have yielded very impressive results, some require a lot of computational resources. In order to make a balanced decision we want to contrast the potential gain in accuracy of some methods with their execution time.

This thesis is structured as follows. In Chapter 2 the background behind different types of word representations is given. In Chapter 3 related work for keyphrase extraction is discussed, and we propose WebEmbedRank, our new method for keyphrase extraction from Web pages. Followed by this, in Chapter 4 various traditional and more recent methods for document classification are discussed. In order to evaluate WebEmbedRank and several classification methods experiments were conducted, which are described along with the setup of the methods in Chapter 5, followed by the results in Chapter 6. Finally, we end the thesis with a final discussion and conclusion about the findings in Chapter 7.

Chapter 2

Word Representations

One of the main challenges for any natural language processing (NLP) task is to determine how to represent text in a computer readable form. Whereas humans have an inherent understanding of language, computers do not. To be able to work with data in a computational sense, text needs to be transformed to a numeric format that supports such processing. More specifically, the standard way to do this is to create a *vector space model* (VSM) of the text [70], turning text sequences into vector representations. Numerous methods have been suggested to perform this transformation. In this section some of the most notable and recently proposed methods are discussed.

2.1 Count Vectorization

One of the most intuitive methods to represent text in a vector space is to consider each unique word as a separate dimension. Each word is then *one-hot encoded* as a sparse vector with the length of the amount of unique words, where only the corresponding dimension of the word is 1. In order to represent a complete text, traditionally the *bag-of-words* (BoW) model is often used, which is a count vector that is the sum of the one-hot encoded word vectors of all words in a document. It lends its name to the fact that it only uses word frequencies, and does not preserve the order of the words. Additionally, no sense of the meaning or context is assigned to the words. For example, the BoW vector for the sentence *"Purchase two large cars"* is completely orthogonal to that of the sentence *"Buy 2 big automobiles"*, while it has the same intention. It furthermore also results in large and sparse vectors, as the dimensionality is determined by the size of the vocabulary. Nevertheless, the BoW embedding is still often used for its simplicity and robustness.

2.2 Word Embeddings

Word embedding methods have been introduced to alleviate the aforementioned problems with one-hot encoding of words. These methods are designed to transform large one-hot encoded vectors to a dense and continuous vector space where contextual relations between words are present. This means that the embeddings of words that are semantically similar are expected to also be close to each other in the vector space. Most state-of-the-art word embedding algorithms use unsupervised methods to create these vectors, based on the notion that *"A word is characterized by the company it keeps"*, as originally proposed by Firth [23]. This suggestion uses the intuition that words that appear frequently in the context of each other are likely to be semantically related. Similarly, words that often occur in the same context are also expected to have similar semantic properties.

The core idea to create models of semantic similarity using the properties observed in a large corpus is not new, and are also known as *distributed semantic models* (DSM). A popular example of a DSM is *latent semantic analysis* (LSA). LSA constructs a matrix of count vectors for a large set of documents from a corpus. By applying singular value decomposition to this matrix, it reduces the dimensionality and captures a generalized semantic model based on the distribution. The resulting model can then be used for several NLP tasks, such as comparing documents based on their semantic similarity. This method later inspired other popular methods, like the topic modelling approach known as *latent dirichlet allocation* [6]. As these models leverage the statistical information of word distributions across documents, they are also called *statistical models*.

The term word embedding was originally coined by Bengio et al. in 2003, who introduced the *neural network language model* (NNLM) to construct a DSM [4]. Contrary to the aforementioned methods, the NNLM was trained to predict words based on a small window of context, instead of factorizing a matrix of statistical information about a document. By doing so this implicitly creates dense and continuous word representations in the hidden layer of the network. This type of approach to create word representations is therefore also known as a *prediction model*.

2.2.1 Word2vec

Although the NNLM was the foundation for many current word embedding techniques, the large shift towards word embeddings in a variety of NLP task largely came after the introduction of Word2vec by Mikolov et al. [62]. A large problem with the NNLM proposed by Bengio et al. was its complexity. Training relatively small 100-dimensional word embeddings for only 5 epochs on the AP News corpus (16 million words) took the authors approximately 3 weeks using 40 CPUs. Instead of making a more complex model to increase the accuracy, Mikolov et al. proposed a simpler model that could feasibly be trained on larger datasets. The neural models they proposed were shallow networks with only a single linear (hidden) projection layer, allowing it to be trained on datasets consisting of more than a billion words. With Word2vec they proposed two new architectures: the *continuous bag-of-words* (CBOW) and *skip-gram* models. With it they also publicly released a toolbox[1] with implementations of their algorithms and models that were pre-trained on large datasets, making it easy to implement in other NLP tasks.

The CBOW variant of Word2vec was trained to predict a word given its context, as shown in Figure 2.1a. The input consists of C one-hot encoded word vectors. As the projection layer is shared for all input words, the input vectors are averaged in the projection layer, hence the name *continuous bag-of-words*. Notable here is the lack of a hidden layer with a non-linear activation function, which according to the authors was the main cause of complexity in the NNLM.

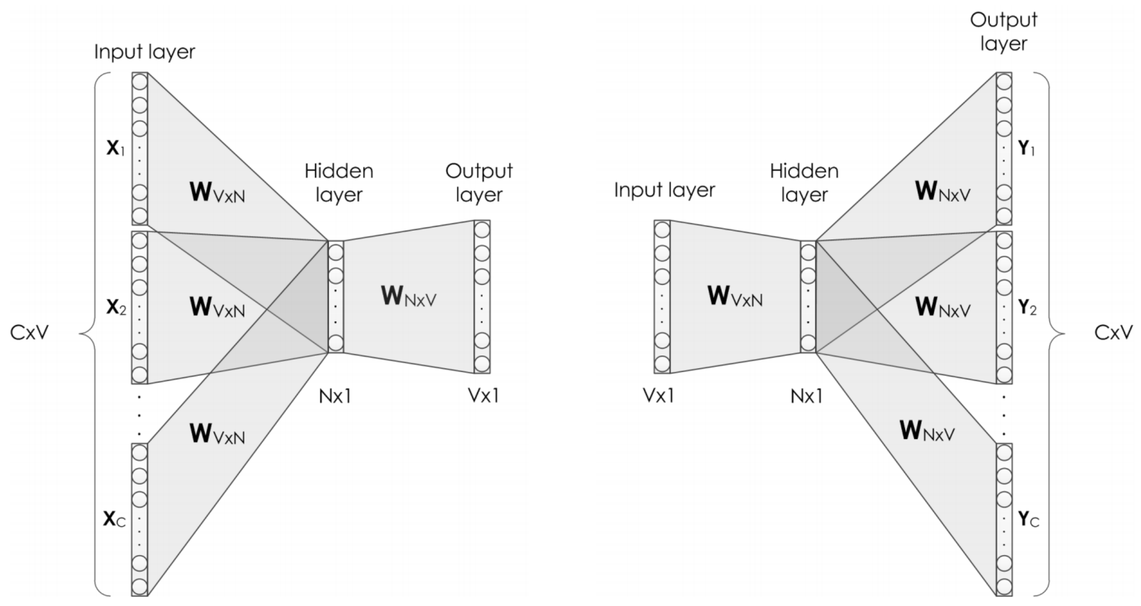
As shown in Figure 2.1b the skip-gram method is essentially a reversed CBOW architecture. Instead of predicting a word given its context, this model is trained to predict the words surrounding a given word. Given a word sequence of size T, the objective function is defined as follows:

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-C\leq j\leq C, j\neq 0}\log p(w_{t+j}|w_t), \tag{2.1}$$

where C is the fixed size of the context window. Most of the tests performed in [62] showed better performance for the skip-gram model, although it also took more than three times longer to train.

In an extension of their original work, Mikolov et al. later proposed modifications to make the Word2vec algorithm more computationally efficient [61]. First of all, they found that the frequent computations of the softmax function on high dimensional vectors was computationally

[1] https://code.google.com/archive/p/word2vec/



(a) CBOW (b) Skip-gram

Figure 2.1: Two Word2vec model architectures as proposed by Mikolov et al. [62] (image taken from [82]).

expensive. To improve on this, they suggested that a hierarchical softmax function can be used instead [63]. The hierarchical softmax uses a binary Huffman tree that can estimate the standard softmax output while being much easier to compute. As an alternative to this, they also proposed to use negative sampling during the training. Usually, the loss is determined based on the output for the complete vocabulary, including all positive and negative samples. With a large vocabulary there are many negative samples, which is the main reason why computing the softmax function is expensive. By only training on the output for all positive samples and a fixed amount of random negative samples, significantly fewer outputs have to be calculated and weights need to be updated each step, strongly increasing the efficiency. As a final improvement, they performed subsampling on frequent terms. Words that occur most frequently in a corpus often contain little semantic value, as these tend to be stop words like *'the'*, *'and'* or *'it'*. By subsampling these words based on an inversed global word frequency, irrelevant yet frequent words are strongly pruned from the dataset while relevant words remain, further reducing the computational complexity.

2.2.2 GloVe

Pennington, Socher, and Manning introduced *Global Vectors* (GloVe) in 2014, arguing that both statistical and prediction based models suffer from drawbacks [67]. They say that statistical methods like LSA make effective use of information from the corpus as they are based on frequency information from the whole document. Unlike this, a context-based method like Word2vec only learns dependencies from local context. However, context-based methods capture the location and order of words in a document. According to Pennington, Socher, and Manning, this can yield a better structured vector space, as shown by its higher performance in the word analogy task. To deal with each of these two problems they introduced the GloVe method.

GloVe attempts to combine the benefits of statistical and prediction models, although it is a statistical model and therefore more like the former than the latter. It works by constructing a global word co-occurrence frequency matrix. The core idea behind their approach lies in the

Table 2.1: Example of word co-occurrence probabilities and ratios for four example probe words, taken from [67].

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

use of word co-occurrence ratios rather than pure frequencies. This can best be illustrated with the example provided by Pennington, Socher, and Manning [67]. First, let X_{ij} be the number of times a word i occurs in the windows of a word j. P_{ij} and X_i are then defined as follows:

$$P_{ij} = P(j \mid i) = \frac{X_{ij}}{X_i} = \frac{X_{ij}}{\sum_k X_{ik}}. \tag{2.2}$$

To illustrate the benefits of considering probability ratios instead of raw occurrence probabilities, consider the terms in Table 2.1. This shows the raw probabilities for the terms *ice* and *steam* in the context of four probe words k, extracted from a 6 billion word corpus. It also shows that only when the ratio is considered, it is easy to find discriminative words for each term by looking how far they diverge from a ratio of 1. For example, *solid* is more discriminative for *ice* as the probability ratio is much greater than 1, and for *gas* and *steam* vice versa. Additionally, it shows that non-discriminative words for both *steam* and *ice* like *water* and *fashion* get scores close to 1.

The above intuition lead Pennington, Socher, and Manning to propose a method that could make use of this information. As shown in the example above, the scores depend on three words: two target words and one context word. This leads to the most general form of the model:

$$F(w_i, w_j, \tilde{w}_k) = \frac{P_{ik}}{P_{jk}}, \tag{2.3}$$

with word vectors $w \in \mathbb{R}^d$ and context vector $\tilde{w} \in \mathbb{R}^d$. The definition of F is devised based on some constraints. First of all, as a vector space consist of a linear structure, F can be constricted to only use the difference between two target words, resulting in:

$$F(w_i - w_j, \tilde{w}_k) = \frac{P_{ik}}{P_{jk}}. \tag{2.4}$$

They furthermore want to avoid F from mixing vector dimensions in non-linear ways, which is prevented by taking the dot product of the arguments:

$$F((w_i - w_j)^T \cdot \tilde{w}_k) = \frac{P_{ik}}{P_{jk}}. \tag{2.5}$$

A problem that now needs to be solved is that it should be invariant to swapping $w \leftrightarrow \tilde{w}$ and also $X \leftrightarrow X^T$, as the difference between a normal and a context word is irrelevant, and can be exchanged. The authors therefore require F to be a homomorphism for the groups $(\mathbb{R}, +)$ and $(\mathbb{R}_{>0}, \times)$. This means that any function $F(A+B)$ with $A, B \in \mathbb{R}$ can be rewritten as $F(A) * F(B)$ while yielding the same results. This also implies that $F(A-B)$ can be rewritten as $F(A)/F(B)$, which allows us to rewrite the lefthand side of Equation 2.5:

$$F((w_i - w_j)^T \cdot \hat{w}_k) = \frac{F(w_i^T \cdot \tilde{w}_k)}{F(w_j^T \cdot \tilde{w}_k)}, \tag{2.6}$$

which can by solved using Equations 2.2 and 2.5, from which can be concluded that $F = \exp$, yielding:

$$w_i^T \cdot \tilde{w}_k = \log(P_{ik}) = \log(X_{ik}) - \log(X_i). \tag{2.7}$$

Finally, the $\log(X_i)$ in Equation 2.7 prevents it from being symmetrical (thus preventing invariance to swapping). As this term does not depend on the context term k, it can be changed to a bias b_i. In order to keep the symmetry, a similar bias $\tilde{b}_k$ is also added, resulting in:

$$w_i^T \cdot \tilde{w}_k + b_i + \tilde{b}_k = \log(X_{ik}). \tag{2.8}$$

While this relatively simple model works, the authors argue that it suffers from treating all co-occurrences equally. Word pairs that only appear together once in the entire corpus are given the same weight as ones that occur very frequently, making it susceptible to noise. To deal with this, they propose to transform Equation 2.8 to a least squares problem, and add a weighting factor in the cost function. Given a vocabulary of size V, they define this new model as follows:

$$J = \sum_{i,j=1}^{V} f(X_{ij})(w_i^T \cdot \tilde{w}_j + b_i + \tilde{b}_j - \log(X_{ij})^2), \tag{2.9}$$

where $f(X_{ij})$ is the weighting function. This function can be defined in several ways, but the authors propose the following:

$$f(x) = \begin{cases} (\frac{x}{x_{\max}})^{\alpha} & \text{if } x < x_{\max} \\ 1 & \text{otherwise} \end{cases}. \tag{2.10}$$

where $x_{\max}$ is a predetermined constant that signifies the cutoff point, and α a parameter that the authors suggest setting to 0.75, which they empirically found to work well. They further show that GloVe can outperform other embedding methods like Word2vec, and can also be more efficient to train than window based methods. However, it is also noted that the complexity of GloVe scales quadratically with the size of the vocabulary. This makes it more efficient for datasets with smaller vocabularies compared to Word2vec, while often being less efficient for large datasets.

2.2.3 FastText

In 2016 a new extension to Word2vec was proposed named *fastText* [7, 42]. Bojanowski et al. argue that previously suggested word embedding methods suffer from limitations raised by ignoring the morphology of a word, as they assign a separate vector to each word. They therefore proposed training the embeddings by breaking up words into character n-grams. Word boundaries are also included (marked by < and > for word start and end respectively), along with the complete word itself. The vector of a word is defined as the sum of the embeddings of all n-grams. For example, the set of n-grams representing the word *'anti'* with $n = 3$ is:

```
{<an, ant, nti, ti>, <anti>}
```

It is important to note the importance of the word boundaries, which in this case make sure that the word *'ant'* with corresponding sequence <ant> is not the same as the *'ant'* in *'anti'*. The authors used all n-grams with $3 \leq n \leq 6$, however suggested that other sets of n-grams could also be considered.

This approach has several advantages. By constructing embeddings from the character n-grams in a word, it can produce relevant embeddings for so called *out-of-vocabulary* words that did not occur in the training set. For example, if a term like *'antigravity'* did not occur during

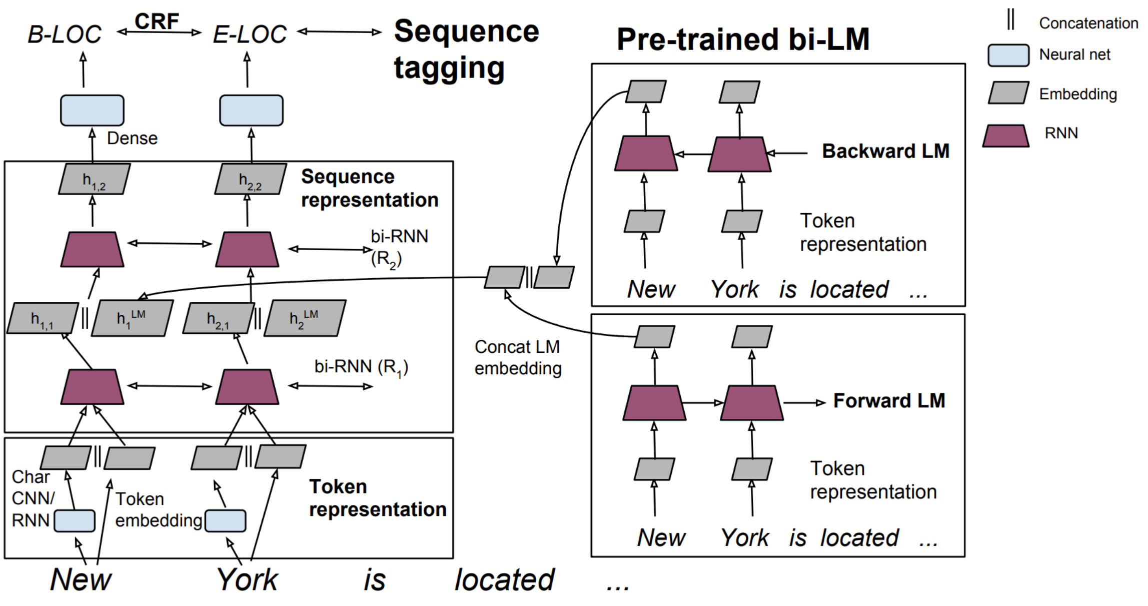


Figure 2.2: The TagLM architecture for creating embeddings from text sequences (image taken from [68]).

training but from the bag of n-grams the words *'anti'* and *'gravity'* are known, it can still create an embedding of it by combining the n-gram vectors. In a similar manner fastText can produce accurate embeddings for misspelled words, slang, or words that occur in various forms or tenses. Additionally, it does not have to learn separate vectors for different morphological forms of the same word. This finally also enables it to produce better embeddings for rare words. Even if a word occurs only a handful of times in a corpus, it is likely that it still shares character n-grams with other, more frequent words. In this way that word can still be assigned a robust embedding.

By using subword information fastText achieved higher accuracies than both the CBOW and skipgram variants of Word2vec on word similarity and analogy tasks. The improvements were most noticeable for languages other than English, such as German and Russian. As these languages feature more declensions, the use of subword information becomes more important. Additionally, German contains many rare words, as multiple words that express one thing are often compounded. For example, the English phrase *'hotel room key'* translates to *'hotelzimmer-schlüssel'* in German. Not treating all declensions and compound words as unique tokens that each required a separate representation therefore significantly increased the accuracy of the embeddings.

2.2.4 Contextual Word Embeddings

Recently the state-of-the-art in word embeddings and NLP in general saw a large shift towards contextualized word embeddings. The main idea behind this approach is that the context a word occurs in can drastically affect what it represents. The previously mentioned embedding methods generate a vector space model where each word in the vocabulary is semantically represented with a fixed vector. However, language is dynamic and polysemic, meaning that words in different contexts do not always have the same meaning. For example, the word *'stick'* has a very different meaning in the sentences *"The dog grabs a stick"* and *"Stick to the plan!"*.

This problem led McCann et al. to propose the Context Vectors (CoVe) method to incorporate context into GloVe embeddings [58]. They did this by leveraging pre-trained sequence-to-sequence translation models. They first trained an encoder-decoder translation model to translate English to German text, using bi-directional LSTMs and an attention mechanism (which are discussed in more detail in Section 4.6). This model is referred to as the MT-LSTM. The representations learned by the model were then used to generate separate context embeddings. The CoVe vectors are created by extracting the output vectors of the MT-LSTM:

$$\text{CoVe}(w) = \text{MT-LSTM}(\text{GloVe}(w)). \tag{2.11}$$

After this the GloVe and CoVe vectors are concatenated to get a final vector for classification tasks:

$$\tilde{w} = [\text{GloVe}(w); \text{CoVe}(w)]. \tag{2.12}$$

By concatenating both vectors, the power of the standard GloVe embeddings trained on billions of tokens is still present, while it also encodes the context it appeared in. Because this process depends on GloVe vectors, it still cannot create embeddings for out-of-vocabulary (OOV) words. McCann et al. suggest to use zero-vectors for OOV words instead.

In 2017 Peters et al. proposed tagLM, a text sequence tagger trained using bi-directional language models [68]. Instead of using the hidden layer from a shallow neural network as the embedding, they proposed using the hidden state from a bidirectional recurrent neural network. First, text is tokenized into single characters. Using either a convolutional or recurrent neural network these tokens are turned into token representations, which are fed into a bi-directional recurrent neural network to form token embeddings. This embedding network is trained on a language modelling task, similar to the prediction task Word2vec is trained on. They then used the resulting embeddings in a sequence tagging model, as shown in Figure 2.2. While this method was effective, it focused on the fairly specific task of sequence tagging.

To provide a more general approach, Peters et al. recently proposed the Embeddings from Language Models (ELMo) architecture. This paper builds on their aforementioned TagLM method, but instead focuses on more general word embeddings [69]. The ELMo method is highly similar to the embedding section of their earlier work, with the main difference from TagLM being that it uses the outputs of all recurrent layers rather than only the final output layer. This means that the resulting word embeddings are created by taking a weighted average over all recurrent layers, which the authors have shown to consistently increase the performance of the model. Similar to CoVe, for classification the final embedding vector is created by concatenating the original token representation created in the first layer with the context embedding from the language model. An advantage of ELMo over CoVe is that it works on a character basis, which means that unlike CoVe it can produce embeddings for OOV tokens. Additionally, the MT model from CoVe requires labeled data to be trained, whereas ELMo can be trained in an unsupervised manner. By using the information from multiple layers of bidirectional language models, ELMo has furthermore shown to be more effective than both TagLM and CoVe.

Chapter 3

Keyphrase Extraction

In order to index and provide summaries of Web pages, it is interesting to see how the most relevant phrases from a page can be extracted. However, doing this is not a trivial task. Web pages are highly inconsistent, and can contain a large number of phrases that are not directly relevant to the page. We will therefore look at several known methods for keyphrase extraction, and how we can adapt these methods to work well with Web page documents.

The field of keyphrase extraction can be split into supervised and unsupervised methods. In the supervised approach the extraction can generally be considered as a binary classification problem, where candidate phrases are classified as either a keyphrase or a non-keyphrase [32]. While they can yield good results and can outperform unsupervised methods [53], supervised methods also come with some difficulties. First and foremost, a supervised approach requires a large set of annotated data, which is very laborious to construct and as far as we know there is no suitable large dataset for this task openly available. Additionally, most supervised methods are domain and language specific [5, 54], while the World Wide Web spans many different languages. This entails that extending a supervised system to another language requires re-annotating the data and training new classifiers for each language. Considering this, we focused on unsupervised extraction methods that can extract keyphrases based solely on the information available on the Web page.

A further distinction can be made between *keyword extraction* and *keyphrase extraction*, although in practice the terms are often used interchangeably. The former consists of finding singular words that can effectively summarize the content of the document, whereas the latter can be a short phrase consisting of multiple words (often referred to as *n-grams*). The advantage of only looking for keywords is that only considering individual words is generally simpler and can be more computationally efficient, as it does not require evaluating all possible combinations of word n-grams. However, single words are often not sufficient in summarizing the core of a document. For example, given a website about San Francisco, both *'San'* and *'Francisco'* are not suitable keywords on their own. Leaving both words out is not desirable either, as they are crucial to describe the topic of the page. As we found the extraction of only single words to be too limiting, we only focused on the extraction of phrases.

3.1 TFIDF

A popular statistical method to determine the importantce of certain words or phrases in a document is to use the *term frequency-inverse document frequency* (TFIDF). This method consists of two parts: the *term-frequency* (TF) and the *inverse document frequency* (IDF). The TF is higher for words that occur more often in the document, and can simply be defined as the ratio of a word w_i in document j. A downside of only using the term frequency to determine importance is that words do not have the same base-rate in terms of how frequently they are used, so using only the TF would lead to common words such as *'the'* to get high scores. This is countered by the IDF part of the equation, which decreases as the amount of documents the word appears in in the corpus increases [77]. Several variants of the IDF scoring exist, however most commonly a logarithmically scaled inversed document count is used:

$$IDF(w_i) = log(\frac{N}{df(w_i)}), \tag{3.1}$$

where N is the total number of documents in the corpus, and df the document frequency, defined as the amount of documents in the corpus in which the word appears at least once. To get the TFIDF score the TF and IDF scores are multiplied. This multiplication effectively leads to a scoring mechanism that gives high scores to terms that appear frequently in one document, but infrequently in all other documents in the corpus.

Although the TFIDF measure is simple and effective for filtering out irrelevant words, it works based on assumptions that may not always hold true for keyphrase extraction. The main assumption that can cause problems is that relevant words do not occur in many other documents. However, some words that occur on many webpages may still be relevant terms. For example, there are many online webshops, many of which will have terms such as *'shopping'* or *'store'* on their homepage. Because of their relatively high occurrence frequency, the TFIDF score decreases for these terms, even though these are relevant keyphrases for this type of Web page.

3.2 Structural Weighting

As mentioned before, Web pages are inherently different from normal text documents, as they are constructed from hypertext (or HTML code). This hypertext contains structural elements that signify how and where parts of the content should be displayed in the rendered document. These elements can therefore provide additional information about the importance of certain segments of texts, which may be helpful in the extraction of keyphrases. For instance, phrases found in the title of a Web page are intuitively more likely to be relevant than phrases taken from a random paragraph on the page. Although the previous research on unsupervised keyphrase extraction from Web pages is limited, it has repeatedly been shown that the HTML element a word or phrase appears in correlates with its relevance for a page [17, 31, 89]. One example of this is the research by Thomaidou and Vazirgiannis, who proposed an unsupervised keyphrase recommendation system for Web pages that used hypertext location information [80]. Their extraction method used a relevance score, which was based on word frequency information with an extra weighting scheme depending on the element a phrase was found in. They based the weights on the locations Web page designers place the most important information on websites, although they provide no further justification for the exact values they chose for the weights.

This domain knowledge has furthermore been used in other NLP tasks using hypertext documents, such as classification. Riboni introduced a weighting method they referred to as the Structure-oriented Weighting Technique (SWT) for Web page classification [72]. The idea behind this weighting was to increase the classification performance by assigning higher weights to texts in elements that were more suitable to represent a Web page, such as the title and meta description. Their results showed higher classification performance when the extra weights were applied, indicating that the title and meta description may contain more important terms than the rest of the page. Similar to this, Kwon and Lee divided elements into three groups of different significance [49]. The highest weighted group contained elements like the title, meta description and headings. These weights were used in combination with a k-nearest neighbor classifier to effectively assign Web pages to several categories.

This notion that the relevance of phrases may vary based on the hypertext elements they appear in is something that can be utilized in the selection of keyphrases. By preferring phrases from elements that have a tendency to be more descriptive for the content on a Web page, we can potentially improve the quality of the selected keyphrases.

3.3 Graph-based Ranking

One of the main subfields in unsupervised keyphrase extraction is the graph-based approach. This idea was first coined by Mihalcea and Tarau, who introduced an algorithm for finding keywords in text based on the PageRank method [65] that they named TextRank [59]. The PageRank algorithm was used by Google to rank websites, representing the websites as vertices in a directional graph that were connected depending on their in- and outgoing links. PageRank can be formally defined as a directional graph $G = (V, E)$ where the set of vertices V are websites and E denotes the set of edges. For a given vertex V_i, $In(V_i)$ is defined as the set of all vertices that point to V_i, and $Out(V_i)$ the set of vertices that V_i points at. In the context of websites this means there is a directed edge from each vertex to all other vertices that vertex contains links to. The PageRank algorithm defines the score or rank of a vertex V_i as follows:

$$S(V_i) = (1 - d) + d * \sum_{j \in In(V_i)} \frac{S(V_j)}{|Out(V_j)|}. \tag{3.2}$$

Here d is a dampening parameter that can be set between 0 and 1, which prevents pages with no outgoing links to act as *'rank sinks'* that accumulate all scores. The authors propose setting this parameter to 0.85. The scores are calculated by iteratively applying this formula until convergence.

With TextRank, Mihalcea and Tarau proposed to adapt the PageRank process to texts by constructing a graph representation of the document, where each word is a vertex [59]. First, all text in the document is tokenized, and part-of-speech (POS) tags are assigned to each token. A filter for certain POS tags can then be applied. The authors recommend filtering out all words that are not tagged as a noun or adjective. The remaining words are represented in a graph where each word is a vertex. The links in the graphs are based on the co-occurrence of words. A moving window of a predetermined size is passed over the text, where two vertices are connected if their corresponding words co-occur in the window. This means that contrary to PageRank, TextRank builds an undirected graph. TextRank makes further adaptations by making the scoring method weighted. The authors argue that in its original domain of ranking websites it is unusual for a website to contain multiple or partial links to other pages, which is why an unweighted graph was used. However, when this algorithm is applied to the word graph it is likely for multiple or partial links to exist. They therefore introduced a weighted version of the scoring method for TextRank:

$$WS(V_i) = (1 - d) + d * \sum_{V_j \in In(V_i)} \frac{w_{ji}}{\sum_{V_k \in Out(V_j)} w_{jk}} WS(V_j). \tag{3.3}$$

Here the new weight parameter w is added, which can be used to add a weight to the edge. This scoring is again applied iteratively until convergence. To avoid making the graph too large by adding all possible n-grams, only single words are considered as candidates. Keyphrases of multiple words are constructed afterwards by first marking all words that are potential keywords, and if multiple marked words are adjacent in the unfiltered original text they are collapsed into one keyphrase.

Several methods have since been proposed that built and improved upon the TextRank method. Wan and Xiao proposed Singlerank [83], which is a slight adaptation of the original TextRank. It sets the weight parameter w for the edges based on the co-occurrence frequency of the words. This ensures that words that occur together more frequently in the document are given a higher weight. They additionally proposed to construct phrases by filtering for noun phrases, and scoring the phrases by summing the scores of individual words to obtain a score for the phrase. While this method showed increased scores over TextRank, Bougouin, Boudin, and Daille suggested that there is no justification for this scoring method [11]. They additionally argued that summing the individual scores to obtain phrase scores leads to longer phrases getting higher scores, even when some of the words in the phrase are irrelevant. To solve this, Bougouin, Boudin, and Daille proposed TopicRank. In TopicRank phrases are first clustered into topics, based on their similarity. The authors suggested placing two phrases into the same cluster if at least 25% of their words overlap. TextRank's ranking model is then applied to a complete graph where the vertices are the topics, in order to assign a score to each cluster. The keyphrases are selected by taking only the most representative candidate from each topic cluster, starting with the cluster that got the highest score. This clustering attempts to avoid overgeneration of similar keywords, however it can not effectively deal with synonyms that are semantically similar, but not syntactically.

Recently, Boudin proposed a variation of TopicRank using multipartite graphs [10], which is often referred to as MultipartiteRank. Arguing that a downside of TopicRank is that all phrases in a single topic are seen as equally important, they proposed to model the topics in a multipartite graph where the vertices are the words, and the partitions the topics. This modification allowed the graph to model the relations between all words, while still being able to represent the different clusters. They also added an additional weighting step, where the edges between words that first occurred early on in the text were given a higher weight.

This positional weighting mechanism was similar to the concept earlier introduced by Florescu and Caragea in the PositionRank method [24]. Their motivation was to include an extra bias based on the location of words, as they argue that important keyphrases are more likely to be used at the start of a text. They do this by adding an extra parameter $\widetilde{p}$, which is a vector containing normalized positional scores for each word. The authors show two ways to calculate this score. In the full model they look at the positions of all occurrences of a given word in a document. For each occurrence a score is calculated as $1/pos(w)$, where $pos(w)$ is the ordinal position of the word. The score p_i for word w_i is then calculated as the sum of the scores for all occurrences of the word. The second version they evaluated is similar, however instead of summing the scores of all occurrences the score is determined only by the first position of the word. Finally, all elements in the vector are normalized:

$$\widetilde{p}_i = \frac{p_i}{\sum_{j=1}^{n} p_j}. \tag{3.4}$$

This normalized positional score is incorporated in the scoring as follows, before any filtering is applied:

$$WS(V_i) = (1-d) * \widetilde{p}_i + d * \sum_{V_j \in In(V_i)} \frac{w_{ji}}{\sum_{V_k \in Out(V_j)} w_{jk}} WS(V_j). \tag{3.5}$$

This bias naturally weighs keyphrases depending on their location in the document. Although Web pages do not always have a well defined order in which elements are read, this bias may prove useful when the text on a Web page is selected in a set order, based on the importance of certain elements. This is something Florescu and Caragea also made us of in lesser extent, as they constructed the document by concatenating the title and abstracts from papers.

3.4 Embedding-based Ranking

With the rise of word embeddings as explained in Chapter 2, new possibilities also emerged for keyphrase extraction. Using the semantic representation of text in the form of vectors can

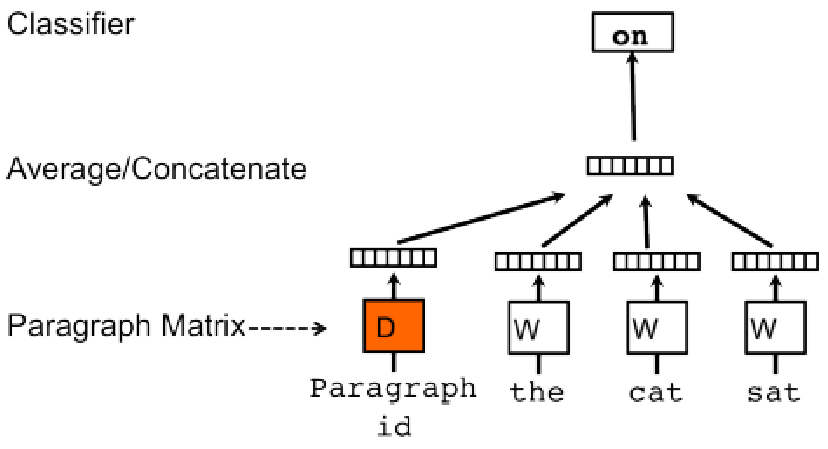


Figure 3.1: PV-DM variant of Doc2vec (image taken from [50]).

be very helpful in determining whether a word is relevant to a document. Recently, Bennani-Smires et al. proposed a new method called EmbedRank that improved on the state-of-the-art of unsupervised keyphrase extraction using text sequence embeddings. They proposed a relatively simple yet effective algorithm that extracts phrases based only on the Web page itself and pretrained sequence embeddings, requiring no corpus. Similar to most other methods, EmbedRank starts with selecting candidate phrases. After tokenizing all text in the document, part-of-speech (POS) tags are assigned to each token. Only phrases that consist of one or more nouns and any adjectives that directly precede them are considered candidates. The candidates are then ranked based on their similarity with the document as a whole. This is done by making word sequence embeddings of the complete document and all the candidates. The similarity between each candidate embedding and the general document embedding is calculated using the *cosine similarity*. This metric is defined as the normalized dot product of the document and candidate embeddings:

$$\mathrm{sim}(p,d) = \frac{p \cdot d}{\|p\| \cdot \|d\|}. \tag{3.6}$$

The candidates with the highest similarity to the document are selected as the keyphrases. EmbedRank therefore ranks the phrases based on how similar, and thus descriptive they are for the complete document.

In the original paper two methods are compared to create the document embeddings. The first is the paragraph vectorization method that has been dubbed Doc2vec [50]. This method is an extension of Word2vec (see section 2.2.1), proposed by the same authors. Doc2vec consists of variants of both the CBOW and skipgram models of Word2vec. However, the authors acknowledge that the former is more accurate than the latter, hence we will only consider the adaptation of the CBOW variant. This model is also referred to as the *Paragraph Vector with Distributed Memory* (PV-DM) model, of which a schematic overview is shown in Figure 3.1. This illustrates how the PV-DM model is similar to Word2vec's CBOW model, with the addition of a paragraph vector D. This vector is unique for each paragraph and is incorporated in the model similar to the other word embeddings. The main difference is that where the word vectors are static across paragraphs, the paragraph vector varies. This allows this vector to act as a memory for the topic of the paragraph, hence the name of the model. Both the paragraph and word vectors are trained using gradient descent and backpropagation, which are explained in more detail in Section 4.4.2. By using document vectors instead of paragraph vectors this paragraph embedding method can easily be generalized to documents. Bennani-Smires et al.

compared the Doc2vec model to Sent2vec [66], which is an extension of the more recently released fastText embedding method (see section 2.2.3). In practice Sent2vec is highly similar to fastText, only it uses word n-grams instead of character n-grams. When only unigrams are considered, the model is trained very similarly to Word2vec's CBOW model, with the main difference being that the context window covers the complete input sentence instead of having a fixed size. Whereas Doc2vec infers document embeddings by feeding all inputs through the network, Sent2vec averages the pre-trained word vectors in a linear pass over all words in the input. This results in an inference complexity of $\mathcal{O}(1)$, meaning that the sentence embeddings can be inferred very efficiently. In the comparison the authors of EmbedRank found that overall Sent2vec outperformed Doc2vec in both inference quality and speed.

The authors furthermore proposed a different version of EmbedRank that incorporates a diversity mechanism to decrease the amount of semantically equivalent keyphrases from being selected, which they refer to as EmbedRank++. This diversity mechanism is an adapted version of the Maximal Marginal Relevance (MMR) metric, which weighs diversity against similarity. It does this by taking into account the cosine similarities between all the candidate embeddings. A parameter λ is used to determine the balance between the relevance (similarity with the document) and the diversity (similarity with other candidates). This parameter can range between 0 and 1, where a higher value indicates an increased importance of the relevance versus the diversity, and vice versa. With a balanced value for λ candidates that are too similar to other candidates but not as relevant are excluded. Although this version scored lower on their evaluation set, they also showed the results of a pilot user study that suggested that users overall preferred the sets of phrases with diversity.

3.5 WebEmbedRank

While the EmbedRank method has been shown to be capable of extracting state-of-the art keyphrases, it does not use the positional information from the text. In the original proposal the method is implemented for extracting keyphrases from scientific documents and news articles. As documents from these domains consist of a single segment of continuous text, there is limited structural information available. When Web pages are treated in the same way, valuable structural information from the hypertext is ignored. To effectively make use of this information we propose WebEmbedRank, an extension to the EmbedRank algorithm that adds an extra weighting scheme to allow it to take advantage of the extra structural information present in Web pages.

The base of WebEmbedRank is similar to EmbedRank. After all text is collected from a Web page, candidates are selected by applying a POS-filter and extracting all combinations of adjacent adjectives, followed by at least one noun. The distance between the text embeddings of the candidate and the document are calculated to get a similarity score. We chose to use Sent2vec [66] to transform the text to embeddings, as Bennani-Smires et al. have shown that it generally worked better and faster than Doc2vec [5]. We furthermore slightly altered the way the document embedding was calculated to deal with the highly varying length of text on Web pages. Instead of basing the document embedding on all candidates, which can sometimes be scarce, the embedding was based on all text on the page after stop-word removal. The stop-words were removed as these yield no semantic value and therefore only added noise to the embedding. While lexical items other than adjectives and nouns may generally not make the best keyphrases, they can contribute to the overall meaning of the page, and we found that using all text improved the performance of WebEmbedRank on our evaluation set.

As the resulting cosine similarity scores can be negative, the similarity scores for all candidates are normalized between 0 and 1 before the weighting. The weight is determined based on the elements a phrase occurs in. As a phrase that appears in multiple key elements is likely

Table 3.1: The weighting scheme for candidate phrases using WebEmbedRank. n is the number of words in the phrase.

Element	Weight
Hostname	4
Title	1.5
Headings	1
Description	0.5
Others	0
Multi-word penalty	-0.25 * (n-1)

to be more important than one that only occurs in one, we propose a cumulative weighting system. This entails that for each candidate phrase, the bonus weight is determined as the sum of all weights of the elements the phrase occurs in. Earlier work has shown that the hostname can be a strong descriptor for a Web page [44, 45], and often consists of the most relevant words for the page such as the brand name, or the main product or service it provides. We therefore included a weight to increase the score for terms that occurred in the hostname. We also included a small penalty for long phrases. As the document embedding is essentially a mean of the word vectors in the documents, a phrase embedding has a tendency to move more towards the mean of all word vectors as the number of words in the phrase increases. For this reason we noticed that EmbedRank had a tendency to generate many long phrases, which may have also been a cause for the lack of diversity described in the EmbedRank paper [5]. By adding a slight penalty to long phrases, we bias the ranking by preferring shorter phrases, while still allowing longer phrases to be generated if the extra words significantly contribute to the relevance of the phrase. The proposed weights are based on an exhaustive grid search on our manually annotated evaluation set. We determined the top-10 keyphrases for all Web pages in the evaluation set for more than 24,000 combinations of weights. The set that achieved the highest average micro-F$_1$ score can be found in Table 3.1, using the annotated keyphrases as golden standard. To illustrate how this bonus weight is calculated, consider a phrase that consists of two words and appears in the title and the description, but not in hostname or headings. In this case the bonus weight is $1.5 + 0.5 - 0.25 * (2 - 1) = 1.75$.

The complete scoring $S(p_i)$ for a candidate phrase is calculated as follows, where $\text{sim}(p_i, d)$ is the cosine similarity between a candidate phrase embedding $p_i \in P$ and the document embedding d, and w_i the bonus weight corresponding to p_i:

$$\widetilde{\text{sim}}(p_i, d) = \frac{\text{sim}(p_i, d) - \min\limits_{p_j \in P} \text{sim}(p_j, d)}{\max\limits_{p_j \in P} \text{sim}(p_j, d) - \min\limits_{p_j \in P} \text{sim}(p_j, d)}, \tag{3.7}$$

$$S(p_i) = (1 + w_i) * \widetilde{\text{sim}}(p_i, d). \tag{3.8}$$

This scoring method allows a sentence that is semantically similar to the complete document to get a significant boost in the ranking if it appears in one or more key locations. At the same time phrases that are highly dissimilar to the document will get a cosine similarity score closer to zero, making the bonus multiplier much less effective. This ensures that semantically representative keyphrases in good locations will get a high rank, while still being robust against irrelevant words that are placed in the important HTML elements. For completeness we also evaluated a WebEmbedRank variant with the diversity preservation mechanism as described in the previous section, which is referred to as WebEmbedRank++. In this case the weighting process (Equation 3.8) is applied before the MMR filtering.

By combining both a-priori knowledge about the importance of specific elements and recently developed text sequence embedding methods, we aim to provide a new method that combines the best of both worlds. WebEmbedRank uses semantic information from the complete Web page, and is strengthened with structural information where available. This combination makes it more robust to the inconsistent nature of Web pages, where it can not be assumed that any of the elements are always present.

Chapter 4

Classification

Classification is the task of mapping an input to one or more discrete categories. To do this, most classification methods are tasked with finding a function $f : \mathbb{R}^n \to \{1, ..., k\}$. In the case of this thesis, we aim to find a function that can predict the category a vectorized text sequence belongs to. These vectors are either high dimensional bag-of-word vectors, or a set of lower dimensional continuous word embeddings. To determine which approach is best suited for this task, various methods that are and have been popular in the field of text classification are compared based on their precision, recall and micro-F_1 score. In this chapter an overview of the chosen methods is given. First, some older yet robust baselines are described. Following this we show more recent developments in text classification such as convolutional neural networks, recurrent neural networks, and attention based networks.

4.1 Logistic Regression

Logistic regression is a popular statistical method for categorical classification problems. It works by fitting a sigmoid (or logistic) function to binomial data, which provides a probability distribution of the two classes. A simple example is shown in Figure 4.1, which shows how a sigmoidal regression curve is fit to a single indicator variable to predict a binomial label. By coding the positive label as *1* and the negative label as *0*, the regression curve yields a probabilistic value of a new sample belonging to the positive class. A classification for that sample can then be made by setting a decision threshold based on the probability, which is usually set to 0.5 for balanced classification. However, if either precision or recall for a class is considered more important, this threshold can be changed accordingly. The previous example showed how logistic regression is applied using a single variable, but in practice input data is often multivariate. In that case a linear combination of the input variables can be used, similar to linear regression. This results in the following equation for the regression curve:

$$\theta^T x^{(i)} = \theta_0 + \theta_1 x_1^{(i)} + ... + \theta_j x_j^{(i)}, \tag{4.1}$$

$$h_\theta(x^{(i)}) = P(y^{(i)} = 1 \mid x^{(i)}) = \frac{1}{1 + e^{-\theta^T x^{(i)}}}. \tag{4.2}$$

The exponent makes it more difficult to determine the optimal values for θ compared to linear regression, which can be trained using the least-squares method. Instead, *gradient descent* is used to optimize the fit of the regression curve. In order to improve the fit of a standard regression curve, we want to change the coefficients θ in a way that minimizes a loss function $\mathcal{J}$. This weight update is calculated using the partial derivative of the loss function in order to find the direction to move the weights in such that it moves towards a local minimum:

$$\theta^{t+1} = \theta^t - \eta \frac{\partial \mathcal{J}}{\partial \theta}, \tag{4.3}$$

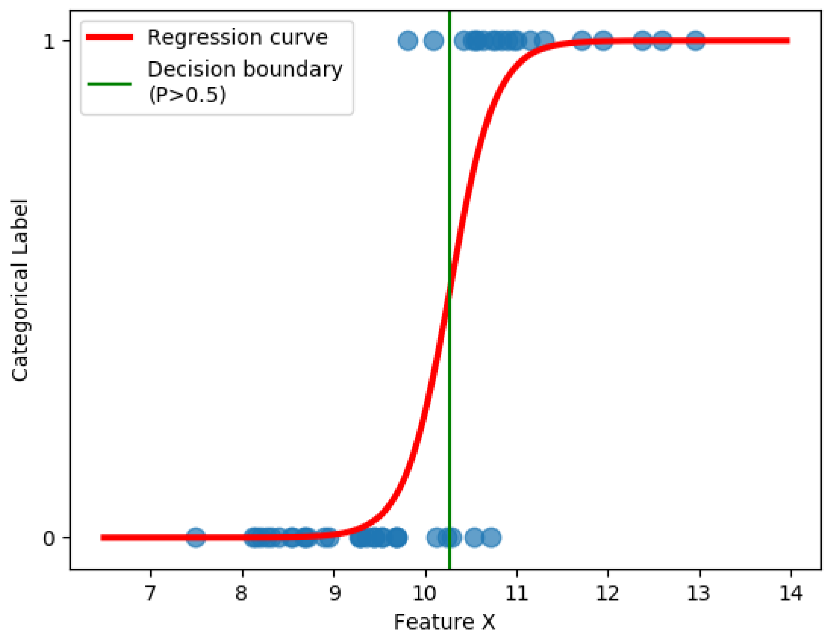


Figure 4.1: Example of a logistic regression curve and decision boundary for binomial data.

where θ are the coefficient parameters, and η is the learning rate. Choosing a suitable value for η is important, as a learning rate that is too low can result in slow training, whereas a value that is too high can make the update overshoot the minimum, resulting in erratic behavior. We also need to determine which cost function $\mathcal{J}$ to use. The mean squared error could be used, like is common with linear regression, however due to the exponent this loss can become non-convex. This means that there can be local minima, which gradient descent may get stuck in. Instead, the *logistic loss* function is often used, which in this case is convex and therefore ensures that the global maximum will be found (given that the learning rate is set correctly). This cost function is defined as follows:

$$\mathcal{J}(h_\theta(x^{(i)}), y) = \begin{cases} -\log(h_\theta(x^{(i)})) & \text{if } y = 1 \\ -\log(1 - h_\theta(x^{(i)})) & \text{if } y = 0, \end{cases} \tag{4.4}$$

where y is the binary label. Since logistic regression is a binary classifier, we know that the label is either 1 (positive) or 0 (negative). This cost function can therefore be rewritten as:

$$\mathcal{J}(h_\theta(x^{(i)}), y) = -(y \log(h_\theta(x^{(i)})) + (1 - y) \log(1 - h_\theta(x^{(i)}))). \tag{4.5}$$

This is the same as Equation 4.4, as the left side is cancelled out if $y = 0$, and the right side when $y = 1$. By taking the derivative of Equation 4.5 the gradient for a parameter j can be defined as:

$$\frac{\partial \mathcal{J}}{\partial \theta_j} = (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}. \tag{4.6}$$

Now that we know the gradient, we can use the gradient descent method from Equation 4.3 to update the weights θ. The aforementioned only assumes updating on a single data point, though in general training is done in batches. This is called *mini-batch gradient descent*, which is further discussed in Section 4.4.2. The update equation for a batch of size m then becomes:

$$\theta^{t+1} = \theta^t - \frac{\eta}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}. \tag{4.7}$$

By iteratively applying these weight updates, the parameters move towards the minimal loss with respect to the input samples. The amount of iterations can be a set amount, or more commonly the training can be stopped when the loss no longer decreases.

Logistic regression is by design a binary classifier, as it yields a single probability . A simple yet effective method for allowing it to be used in multinomial settings is to use the *one-vs-rest* approach. Given a classification problem with K classes, the one-vs-rest method trains K separate binary classifiers - one for each class. For each class i a classifier f_i is trained using all data belonging to class i as positive samples, and all others as negative samples. After training new samples are classified using $f(x) = argmax_i f_i(x)$.

4.2 Random Forest

The random forest classifier is an ensemble learning technique that is based on decision trees. A decision tree is a model that is made by learning simple decision rules based on the provided features. Decision trees are generally generated in an iterative top-down process, where consecutive tree splits are determined based on the feature that can best split the data at that point. To determine what constitutes the best split, various metrics have been proposed to calculate the impurity of a split. In general, the impurity is based on the distribution of classes among the nodes after the split. If the nodes after a split each contain many samples of only one class, the impurity is low, which indicates a discriminative split. One of the more popular metrics for finding the impurity is the entropy:

$$i(t) = -\sum_{j=1}^{k} p(j|t) \log_2 p(j|t), \tag{4.8}$$

where k is the numbers of classes, and $p(j|t)$ the fraction of data points from class j in node t. Another method that is frequently used is the Gini impurity, which is slightly less computationally expensive to compute as it does not contain a logarithm:

$$i(t) = 1 - \sum_{j=1}^{k} p(j|t)^2. \tag{4.9}$$

One of the differences between these two methods is that for two classes entropy ranges from 0 to 1, whereas Gini ranges from 0 to 0.5. Both are measures of impurity, so splits with lower impurity are in this case considered better. The *information gain* of each split is calculated for each split by subtracting its impurity from the impurity before the split. The option that yields the highest information gain is then chosen. This is applied in an iterative process that can continue until all nodes are pure, however doing so may lead to overfitting, especially when many features are considered. A common way to prevent this is to set a maximum tree depth.

One of the most prominent advantages of decision trees is that they are easy to understand, as the resulting decision tree can be visualized to explain how the decisions are made. Additionally, thanks to its simplicity the computational cost to train a decision tree is low. However, a common problem is that decision trees have a tendency to overfit, and are known to be unstable as small variations in the training data can completely change the tree that is generated [78]. Random forests attempt to overcome these downsides through an extension of the *bagging* method. Bagging is an abbreviation of *bootstrap aggregating*, which is a method where multiple weak learners are trained on subsets of the data, and combined in an ensemble. This process has shown to improve unstable methods like decision trees by reducing the variance of a prediction function [12]. Random forests extend this method by not only training the estimators on random samples of the data, but also by using random subsets of the features for each node split. This further improves the variance reduction, leading to more stable results [33]. A final classification is made by a majority vote of all the trees in the ensemble.

While a random forest classifier can often produce more consistently good results, it does lose the interpretability that decision trees offer. In general, the stability and overall performance becomes better as the number of trees in the ensemble increases. However, including more trees also increases the computational complexity. Depending on the problem, usually somewhere in the tens to hundreds of trees are used.

4.3 Support Vector Machine

A *support vector machine* (SVM) is a classifier that attempts to optimally separate two classes in a vector space [20]. When the data is linearly separable, the SVM aims to find an n-dimensional hyperplane defined as $w * x + b = 0$ that provides the largest margin between the hyperplane and the closest data points from each class, which are also known as the *support vectors*. The two classes are labeled -1 and 1, which means we are looking for a decision boundary that satisfies the following constraints for all labels y:

$$w * x_i + b = \begin{cases} \geq 1 & \text{if } y_i = 1 \\ \leq -1 & \text{if } y_i = -1, \end{cases} \tag{4.10}$$

which can be rewritten in a single line as:

$$y_i(w * x_i + b) \geq 1, \quad \forall_i. \tag{4.11}$$

In a linearly separable case, there are infinite possible hyperplanes that satisfy these constraints. What we want to find is the hyperplane that can separate the two classes with the largest margin, also known as the *maximal margin hyperplane*. To do this we want to minimize the following function subject to Equation 4.11:

$$\frac{1}{2}||w||^2, \tag{4.12}$$

which can be written as a Lagrangian function with Lagrangian multipliers α_i:

$$L = \frac{1}{2}||w||^2 + \sum_{i=1}^{N} \alpha_i(1 - y_i(w * x_i + b)). \tag{4.13}$$

However, this assumes that the constraint applies that the data is linearly separable. In practice, datasets are often not linearly separable, which the SVM addresses with two methods: *soft margins* and the *kernel method*.

The addition of *soft margins* makes it possible to have a decision hyperplane that does not perfectly separate all data points from the training set. By introducing a penalty for each data point on the wrong side of the decision boundary instead of disallowing them completely, the conditions for the separating hyperplane are relaxed. This is implemented by adding the slack variables ξ, and a penalty parameter C as follows:

$$\frac{1}{2}||w||^2 + C\sum_{i=1}^{N} \xi_i + \sum_{i=1}^{N} \alpha_i(1 - \xi_i - y_i(w * x_i + b)). \tag{4.14}$$

Here C controls the trade-off between getting the maximum margin between the support vectors and the degree to which misclassification is allowed. A low value for C allows for a large margin that may misclassify many samples, whereas a high value for C puts more weight on classifying all samples correctly which can in turn cause the decision hyperplane to overfit on the training data.

Additionally, the *kernel method* (also known as the *kernel trick*) is used to map the data to higher dimensions in which it is linearly separable. These extra dimensions are created by kernel functions that use the inner products of pairs of features. In the resulting implicit feature space the linear classifier can be used, allowing for complex non-linear decision boundaries in the input space. Several kernel functions can be used, such as a linear, polynomial or radial-basis kernel, with varying computational complexities. Because text represented in the bag-of-words format is already very high-dimensional, it lends itself well to classification with an SVM. This high dimensionality also makes the input data likely to be linearly separable, causing the much more computationally efficient linear kernel to often be sufficient for good classification [41].

In a similar fashion to logistic regression, the SVM is made to perform binary classification. Likewise, for SVMs multi-class classification can also be implemented using one-versus-rest classification (see Section 4.1).

4.4 Multilayer Perceptron

The multilayer perceptron is a feed-forward artificial neural network, and is considered as one of the most basic forms of neural networks. As the name suggests, it is based on multiple layers of *perceptrons*. A single perceptron is a linear classifier loosely based on the workings of individual biological neurons [73]. It takes multiple inputs, which are turned into a single output using a linear combination of the inputs. Often a non-linear activation function is applied to the output. This can be represented as follows:

$$y = \varphi(\sum_{i=1}^{N} x_i w_i + b). \tag{4.15}$$

Here N is the number of inputs, x is the input vector, w the weight vector, b the bias and φ the activation function. In multilayer perceptrons layers of these perceptrons sequentially feed into each other, where the perceptrons in each layer are fully connected to all perceptrons in the neighbouring layers. To be considered a multilayer perceptron, there need to be at least three layers. These consist of an input layer, one or more hidden layers and an output layer.

4.4.1 Activation Functions

The non-linear property of the activation function is important, as a combination of linear functions will always result in another linear function, losing all benefits of combining multiple layers. By having a non-linear activation function, the network is able to generate more complex non-linear decision boundaries. In the original perceptrons the activation function φ is a step-function where the output is 0 if the value is smaller or equal to 0, and 1 when it is larger than 0. In this case the bias is trained and used as the threshold value that determines when a neuron fires. However, in order to train a neural network it is useful to be able to make small changes that gradually improve the network. Therefore continuous activation functions are more suitable for multilayer perceptrons. Traditionally two activation functions have been very popular, being the sigmoid and hyperbolic tangent (tanh) function. The sigmoid is the same as used in logistic regression (see Section 4.1) and is defined as follows:

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}}. \tag{4.16}$$

The sigmoid bounds the output between 0 and 1 in a non-linear way. The tanh function is defined as:

$$\tanh(x) = \frac{2}{1 + e^{-2x}} - 1. \tag{4.17}$$

As the equation shows, the tanh function shows clear similarities to the sigmoid function, which is because it essentially is a scaled variant of the sigmoid. Instead of ranging between 0 and 1, the output of the tanh function ranges between -1 and 1. This makes it have a stronger gradient during training as the derivatives are steeper. However, recently a much simpler activation function known as the ReLU has been gaining popularity:

$$\text{ReLU}(x) = max(0, x). \tag{4.18}$$

The Rectified Linear Unit, commonly referred to as ReLU, has quickly been adopted by many researchers and has currently become the most widely used activation function [71]. The main reason for this can be found in both its simplicity and effectiveness. The ReLU allows networks to be trained faster by decreasing the effects of vanishing gradients, which are further discussed in Section 4.4.3.

The exact architecture of the multilayer perceptron can be optimized for different tasks. For multi-class classification problems it is common practice to have one node for every class in the output layer. The *softmax* function is often applied to the output layer [13], which can be defined as follows:

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}. \tag{4.19}$$

Here the output score is calculated for output node z_i, where K is the number of output nodes. As the name suggests, this provides a softer version of a max function. Instead of giving a hard classification to the highest scoring node, it calculates a probability distribution over all output nodes, allowing for a probabilistic interpretation of the output. To compare this type of output with the categorical labels, the labels are generally *one-hot encoded*. This encoding produces a sparse vector with one dimension for each class, where only the dimension that corresponds to the label is 1.

4.4.2 Training

The network is trained by updating the weight vectors of each layer in a process called gradient descent. Initially the weights of the network are set randomly. Then, given a training sample, a *forward pass* is made by feeding the sample through the network, making a first prediction. This prediction is compared to the desired output using a *loss function* to calculate the *loss* or *error* of the given sample. Various loss functions have been proposed for different models and purposes, but for multinomial classification problems with a softmax output layer and one-hot encoded labels it is common to use the *multinomial cross entropy* (also known as the *negative log likelihood*):

$$\mathcal{L}(x^{(i)}) = -\sum_{k=1}^{K} y_k^{(i)} \log(\hat{y}_k^{(i)}), \tag{4.20}$$

where y is the actual label, and $\hat{y}$ the predicted label probability. As the labels are one-hot encoded, y acts as an indicator function. The cross-entropy loss exponentially increases as the predicted label deviates from the actual label, heavily penalizing predictions that are confident yet wrong.

In order to improve the network, we want to change the weights in a way that minimizes the loss. To effectively do so gradient descent is used, similar to Logistic Regression (Section 4.1). Just like before, gradient descent takes small steps towards a local minimum by finding the derivative of the loss function with respect to the weight parameters and iteratively moving in the downwards direction. However, in MLPs the output is dependent on several embedded functions. For example, consider a simple single hidden layer MLP with an arbitrary activation function A and loss function $\mathcal{L}$. In order to find the loss for one sample given the outputs of the hidden layer H, we calculate the following:

$$\mathcal{L}(A(Z(WH))), \tag{4.21}$$

where Z is the weighted input. To find the partial derivative of the loss function with respect to the weights we need to use *backpropagation* [74]. Backpropagation iteratively applies the chain rule such that the gradients can be calculated in an efficient way. This enables us to find the gradient for a weight w_i as follows:

$$\frac{\partial \mathcal{L}}{\partial w_i} = \frac{\partial \mathcal{L}}{\partial A} \cdot \frac{\partial A}{\partial Z} \cdot \frac{\partial Z}{\partial w_i}. \tag{4.22}$$

During training we know the output A for a given sample, as this is the output of the forward pass $\hat{y}$. The gradient for the second set of weights w_i^2 in a single layer neural network can therefore be calculated as follows:

$$\frac{\partial \mathcal{L}}{\partial w_i^2} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial Z_i^2} \cdot \frac{\partial Z_i^2}{\partial w_i^2}. \tag{4.23}$$

The gradient of the first set of weights is dependent on the layers after it, so if we want to calculate it for a weight w_i, we need to consider the output of all nodes in the following hidden layer H. We can again apply the chain rule to calculate the gradient:

$$\frac{\partial \mathcal{L}}{\partial w_i^1} = \Big(\sum_j \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial Z_j^2} \cdot \frac{\partial Z_j^2}{\partial H_j^1}\Big) \cdot \frac{\partial H_i^1}{\partial Z_i^1} \cdot \frac{\partial Z_i^1}{\partial w_i^1}. \tag{4.24}$$

With multiple hidden layers, this chain rule can simply be applied iteratively to find the gradient of the weights in all layers. However, there is much redundancy in these calculations. Given an MLP with three hidden layers, the calculation of gradients for the second hidden layer require the derivatives of the loss function with respect to the weights of the third hidden layer. Consequently, to find the gradients of the first hidden layer we need to calculate the gradients of the second and third layer. Instead of recalculating the gradients for each layer for all weights, back propagation starts calculating the gradients at the last layer. The resulting gradients are then stored, and used in the calculation of the penultimate layer. This process then continues for all layers, propagating backwards through the network to efficiently calculate the gradients. When these gradients are known, the weights and biases are adjusted in the negative direction of the gradient to decrease the loss for the given training samples.

The process of standard gradient descent is very slow, as for every iteration it needs to evaluate and calculate the gradients of all samples in the set, after which the weights and biases are updated. Seeing as neural networks are often applied to big data problems with millions of samples, this quickly becomes problematic. A variant that converges faster is called *stochastic gradient descent*. Instead of calculating all gradients, this method randomly selects one example from the data set, based on which the weights and biases are updated. This results in faster training as weights are updated more often. However, because the gradient of just a single sample is used the training tends to become erratic. In order to strike a balance between these two extremes, it is standard practice to use *mini-batch (stochastic) gradient descent* or variants thereof. This works by dividing the input data into subsamples of a predetermined size, called mini-batches. By having each training step update based on a mini-batch, a balance is struck between efficacy and efficiency. The optimal size of the mini-batches is dependent on the data, model architecture, and hardware that is used, and is often found empirically. Two popular optimization variants are RMSprop [81] and Adam [48]. RMSprop can make the learning more robust by dividing the learning rate with an exponentially decaying running average of the squared gradients for the current parameter [81]. Adam uses a similar idea of having custom per-parameter learning rates, and is sometimes considered as an extension of RMSprop. Apart from using the squared gradients to determine the learning rate, it also incorporates an exponentially decaying running average of past gradients (similar to stochastic gradient descent with momentum) [48].

4.4.3 Vanishing & Exploding Gradient Problem

A large difficulty that comes with the use of gradient descent and backpropagation in artificial neural networks is that of the *vanishing gradient problem* [36]. As explained before, backpropagation works by moving backwards through the network, computing the loss-gradients of the layer with respect to the layer's weights. However, the derivative of traditional activation functions like the sigmoid (Eq. 4.16) can be small, and can quickly move towards zero for both high and low input values. To propagate the error through layers of the network, these small derivatives are multiplied together. This process causes the updates to decrease exponentially with each hidden layer in the network. For networks with few layers this is not a large problem, but the state-of-the-art is moving towards very deep neural networks, which in some extreme cases can consist of more than one thousand layers [34]. Using the traditional methods described above, the gradient will have almost completely vanished after a few layers, which results in the initial layers of a deep network hardly or not being trained.

This problem in combination with the recent trend of using increasingly deep neural networks is the main reason for the quick adoption of the ReLU activation function (Eq. 4.18). Because the ReLU has no upper bound, the derivative does not plateau for higher input values, diminishing the vanishing gradient problem. Additionally, it introduces sparsity in the network as all neurons with zero or lower output do not fire, which has been argued to provide various benefits [25]. It may be noted that the ReLU is technically not suited as activation function as it is not always differentiable. When the input is zero, the ReLU has no derivative. In practice this is often solved by simply setting the derivative of a zero-input to 0, making the derivative 1 for $x > 0$, and 0 otherwise.

The opposite of the vanishing gradient problem is known as the *exploding gradient problem*. This can happen when the gradients become larger than 1. In this case the gradient can increase exponentially when it is propagated through the network, causing it to '*explode*'. Consequently, this can result in erratic behavior during the training of the network. This problem can often be avoided by using batch normalization [39], in which the inputs of each layer are normalized for every mini-batch, or by clipping the gradients.

4.5 Convolutional Neural Networks

The classifiers mentioned in the previous sections all take one input for a complete document (often a bag-of-words vector), and use this to make a prediction. However, for text classification there is one big downside to this approach: they do not take the sentence structure into account. In order to understand the intention of a sentence, it is often not enough to only know which words are in a sentence and how often they occur. Consider the sentence *"We only sell to businesses, not to consumers"*. Now take the following sentence, with a clearly reversed intention: *"We only sell to consumers, not to businesses"*. When using a count-based method, these two sentences are treated identically, which can cause errors in classification.

Convolutional Neural Networks (CNNs) are a type of feed-forward deep neural network that contain one or more layers that apply a convolution to a structured input. A convolution works by passing a *kernel* or *filter* over the input data. These kernels have a fixed size, known as a receptive field. By moving the kernel over each input location and calculating the dot product between the receptive field and the kernel weights, a feature map is created. During training of a CNN the weights of the kernel are optimized to find relevant patterns, allowing it to encode topological information. These convolutional layers are often followed by a *pooling* layer, which subsamples the feature map for better generalization. A popular variant is the *max-pooling* layer, which subsamples by outputting only the maximum value in a window of a fixed size. Its first appearance in machine learning was in a paper published at the end of 1989 by LeCun et al., who used a small CNN for handwritten postal code detection [51]. Their inspiration for this was biologically inspired, following research on the neurons in the visual cortex of a cat [37, 57]. It is therefore no surprise that CNNs are mainly used in fields that deal with computer vision tasks.

The usage of convolutions is however not restricted to image related tasks, as this operation can be used for any task that can be interpreted as a sequence. This also includes texts, as a text consists of a sequence of words (which in turn consist of a sequence of letters). By tokenizing a text sequence and representing each token as a k-dimensional word embedding, a text can be represented as a 2-dimensional matrix by concatenating the embeddings. This representation enables the use of convolutional operations, although usually only 1-dimensional convolutions are used. As there is no meaningful order to exploit from the k dimensions in a single word embedding, the width of the kernel is usually defined as k, such that the convolution only happens on the dimension of the tokens.

4.5.1 C-CNN

A variant of the normal CNN for text classification was developed by Kim [47], that was based on networks earlier proposed by Collobert et al. [19]. The proposed architecture is relatively simple, containing only a single convolutional layer. An interesting addition and key part of this network is that their convolutional layer contains multiple filter widths. The input text is first tokenized, after which each token is transformed into a k-dimensional embedding vector v_i. The input vector is created by concatenating the vectors: $x = [v_1 \oplus v_2 \oplus ... \oplus v_n]$, where $\oplus$ is the concatenation operator. A convolution is applied to obtain a feature c_i for a window $x_{i:i+h-1}$ as follows:

$$c_i = f(w \cdot x_{i:i+h-1} + b). \tag{4.25}$$

Applying this operation to all possible windows yields a feature map $c = [c_1, c_2, ..., c_{n-h+1}]$. A problem with using layers with multiple filter widths is that it results in feature vectors of varying sizes, which is hard to deal with in following layers. This problem is solved by adding a *max-over-time pooling* layer after the convolutional layer. This pooling layer takes a feature map c from a convolutional filter, and applies max-over-time pooling as $\hat{c} = max\{c\}$. Whereas conventional 1-d max-pooling layers reduce dimensionality by taking the max value from strides over the feature map, the max-over-time pooling method used here takes the maximum value from the entire feature map. This means that each feature map will be downsampled to one value regardless of the amount of input values, allowing it to naturally deal with the variability

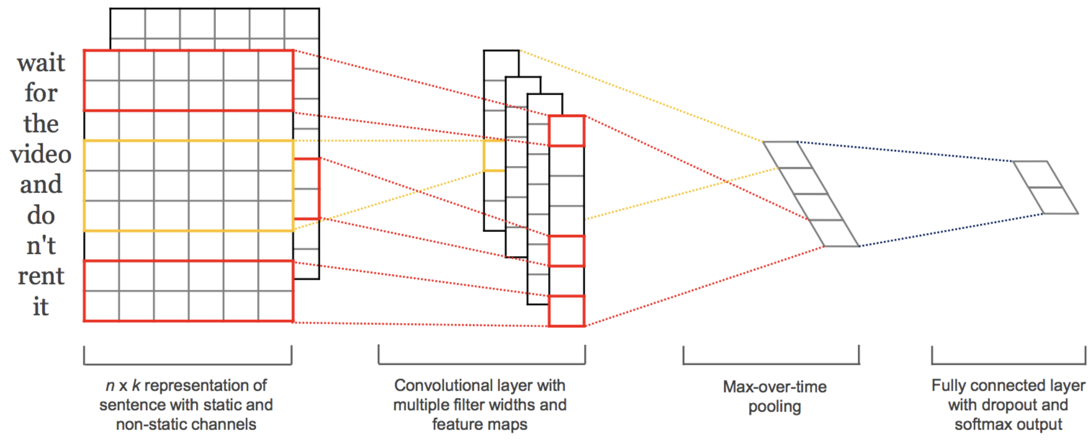

wait
for
the
video
and
do
n't
rent
it
n x k representation of sentence with static and non-static channels
Convolutional layer with multiple filter widths and feature maps
Max-over-time pooling
Fully connected layer with dropout and softmax output

Figure 4.2: Convolutional neural network for text processing with multiple filter widths (image taken from [47]).

in the length of sentences and feature maps. The outputs of the max-over-time pooling are concatenated to form a feature vector of a fixed length, which is why this architecture is referred to as the *Concatenation-CNN* (C-CNN). A schematic overview of the architecture is shown in Figure 4.2.

Kim furthermore experimented with four different input methods. Their baseline model randomly initialized the word embeddings, which were updated during the training. Two further models were initialized with pre-trained word embeddings from Word2vec. For one of these models the Word2vec embeddings remained unaltered, while for the other model the weights were fine-tuned during training. These methods were respectively called the *static* and *non-static* models. The final model, referred to as the *multichannel* model, used two sets of word embeddings in two channels. One channel was static, while the other was trained. The idea behind this was to keep the power of fine-tuning the embeddings for specific tasks, while avoiding overfitting. The results of these four models on several datasets were mixed, although the use of pretrained word embeddings largely increased the scores. The non-static and multichannel models furthermore scored slightly higher on average than the static model, although they were not consistently better across all tasks.

4.5.2 Character-level CNN

Instead of looking at text sequences at the word level, it is also possible to treat it as a sequence of characters. Zhang, Zhao, and LeCun argue that convolutional neural networks are useful for extracting information from raw, low level signals [88]. By moving from the word to the character level, the convolutional network treats text as a raw signal. Doing this can simplify the engineering process of building a classifier, as it requires no understanding of semantics of words, and unlike word level networks it requires little to no preprocessing such as word tokenization. They further argue that a character-level CNN can account for misspellings of words, and other symbols that are otherwise seen as out-of-vocabulary or removed such as emoticons.

In order to vectorize the text sequence, a set of accepted characters had to be determined. All characters that were not present in this predetermined dictionary were dropped. The dictionary used by Zhang, Zhao, and LeCun contained the following 70 characters:

```
abcdefghijklmnopqrstuvwxyz0123456789
-,;.!?:’’’/\|_@#$%^&*~‘+-=<>()[]{}
```

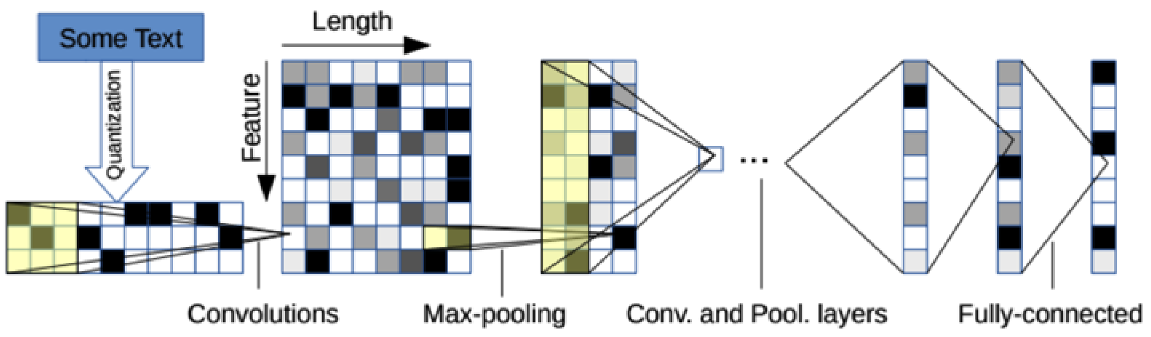


Figure 4.3: Character level convolutional neural network proposed by Zhang, Zhao, and LeCun (image taken from [88]).

They also experimented with differentiating between capitalized and normal characters, however they found that not making this distinction produced better results. Each character in the input was then vectorized using one-hot encoding. The length of each document was limited to the first 1,014 characters, resulting in an input tensor of 1,014 by 70 per document.

Figure 4.3 shows an overview of the architecture used by the Char-CNN. The first two layers were convolutional layers with a kernel size of 7, followed by a max-pooling layer of size 3 with a stride of 1. The four following layers also were convolutional layers, but with a kernel size of 3. Of these four, only the last was followed by a max-pooling layer, again of size 3. The final three layers were fully connected layers, with the last being a softmax output layer. Two dropout layers were added between the fully connected layers, with a dropout percentage of 50%. The authors furthermore compared a big and smaller model. For the big model the size of the convolutional and fully connected layers were 1,024 and 2,048 respectively, while being 256 and 1,024 for the smaller model.

The Char-CNN showed good results compared to baselines, although only for very large datasets. For datasets smaller than a million samples it did not perform as well, often being beaten by relatively simple baselines. They furthermore showed mixed results for the large and smaller model, with neither consistently outperforming the other.

4.6 Recurrent Neural Networks

A *Recurrent Neural Network* (RNN) is a type of neural network that is designed to model sequential data. All neural architectures described so far were feed-forward networks that contained no cycles. Contrary to this, an RNN contains feedback loops in the form of recurrent connections. To visualize the recurrent connections, they are often unrolled over time, which we will also do in this thesis. In this case each timestep has its own layer that feeds into the next in the sequence, while sharing the weights between layers. As the amount of layers can vary without changing the output (depending on which outputs are used), they can naturally deal with sequential input, even when the length of the input varies.

A standard RNN starts with an initial hidden state h^0. For each sequential step t the activation is calculated by combining the weighted hidden state of the previous timestep h^{t-1} with the new weighted input x^t:

$$a^t = Wh^{t-1} + Ux^t + b. \tag{4.26}$$

This is then used to updated the hidden state for the current timestep:

$$h^t = \tanh(a^t). \tag{4.27}$$

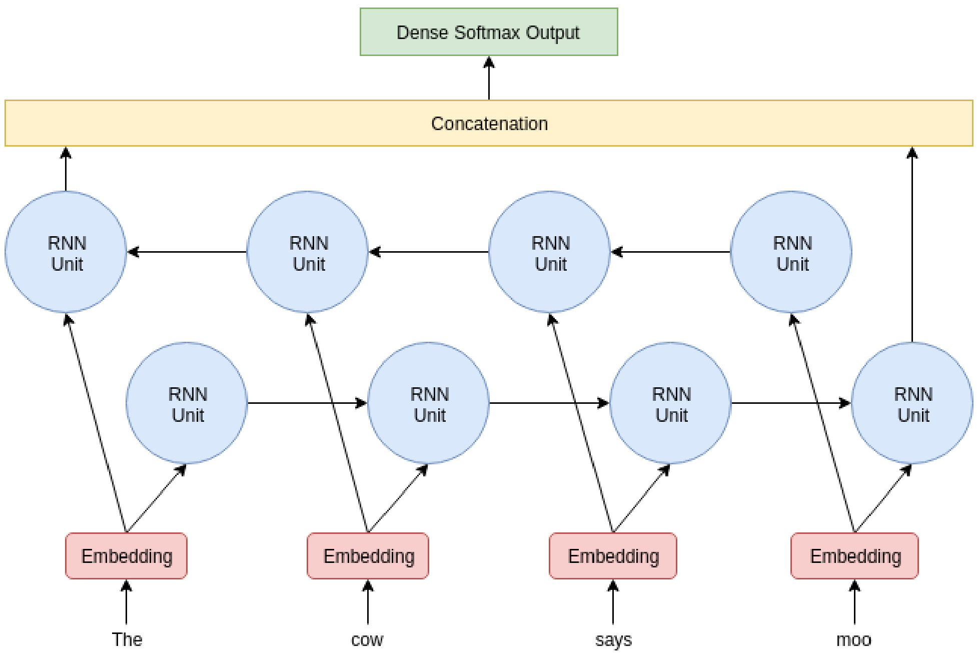


Figure 4.4: Schematic overview of a standard bidirectional RNN for text classification.

This new hidden state is used to calculate a weighted output, which yields the output $\hat{y}^t$ for timestep t after applying a softmax function:

$$o^t = Vh^t + c, \tag{4.28}$$
$$\hat{y}^t = \text{softmax}(o^t). \tag{4.29}$$

Here b and c are the bias vectors, and U, W and V the weight matrices. The *tanh* and $softmax$ functions are the same as defined in equations (4.17) and (4.19) respectively. As is clear from these equations, the RNN takes the state from a previous timestep, and combines it with next input from the sequence. This allows it to dynamically learn from sequential data. RNNs can be trained like other neural networks with gradient descent, where the gradients can be calculated with a slight variation of the aforementioned backpropagation algorithm called *backpropagation-through-time* or BPTT [84, 64].

4.6.1 Bidirectional RNNs

In traditional RNNs the information only flows in one direction, which makes sense when only past information is known and a following state needs to be predicted. However, when we try to classify documents we get access to the complete text at once, and therefore potentially do not use all information by only moving in one direction. This can happen because in texts something that is mentioned later on can be relevant for what happened before. For example, when we want to predict which word should filled into the blank in the sentence *"The ___ says moo"*, we know that it should be the word *'cow'* from the extra information in the context after the blank.

To use this information, Bi-directional RNNs (B-RNNs) were introduced by Schuster and Paliwal, which can utilize context in both directions [76]. A B-RNN adds an extra layer of recurrent units that moves backwards through time parallel to the original forward moving layer. As illustrated in Figure 4.4, the two directional layers of units have no direct connection to each other. Though it is possible to sum or take the mean of the two unit outputs to reduce dimensionality, usually the two outputs are concatenated so that no information is lost. B-RNNs have earlier been found to outperform uni-directional RNNs [29, 87], and have been widely adopted in sequence classification tasks where both past and future context is available, such as handwriting recognition [30], speech recognition [28] and text-to-speech systems [1, 22]. In most state-of-the-art models the units are generally gated units such as the LSTM or GRU, which will be explained in more detail in the following two sections.

4.6.2 Long Short-Term Memory

In theory a standard RNN is able to learn long term dependencies from sequential data. However, in practice it has been shown that just like the multilayer perceptrons it can suffer from the vanishing gradient problem (see section 4.4.3). This makes it practically unable to successfully learn dependencies from sequences larger than 10 elements [2, 3]. To deal with this problem, Hochreiter and Schmidhuber developed a gated variant of the standard recurrent architecture called the *long short-term memory* (LSTM) [35].

The core idea behind the LSTM lies in the addition of the cell-state C. The cell state can also be seen as the memory of the cell, which can at each step be updated using two gates: the *forget gate* and *input gate*. Additionally a third gate called the *output gate* is used to determine the output and update the hidden state. These gates make it easy for information to flow through the network unchanged, allowing it to form long-term dependencies. The forget gate f determines what should be forgotten from the cell state. To update it with new information, the input gate i determines what values should be updated in the cell state. Finally, the output gate o determines how much of the cell state values should be used in calculating the output activation. These gates are defined as:

$$\begin{aligned} f_t &= \sigma_s(W_f x_t + U_f h_{t-1} + b_f), & (4.30)\\ i_t &= \sigma_s(W_i x_t + U_i h_{t-1} + b_i), & (4.31)\\ o_t &= \sigma_s(W_o x_t + U_o h_{t-1} + b_o), & (4.32) \end{aligned}$$

where σ_s is the sigmoid activation function that outputs a value between 0 and 1. As this shows, the equations for each gate are very similar, but they each have their own weights and biases. The cell state is updated by first multiplying the previous cell state with the output of the forget gate. As the output of the forget gate is a value between 0 and 1, this number determines how much is forgotten. This is followed by adding the product of the input gate activation and the new candidates for the cell state:

$$C_t = f_t \circ C_{t-1} + i_t \circ \sigma_h(W_c x_t + U_c h_{t-1} + b_c), \tag{4.33}$$

where $\circ$ is the element-wise multiplication of two vertices, also known as the Hadamard product, and σ_h the hyperbolic tangent (tanh) activation function. Finally, the new hidden state is calculated using the output gate activation and the new cell state:

$$h_t = o_t \circ \sigma_h(C_t) \tag{4.34}$$

As at each timestep the gates determine what should be kept and what should be forgotten, it is possible for information from far away timesteps to remain in the cell's memory. This can additionally decrease the risk of exploding gradients [26], although it is not guaranteed to prevent it [79].

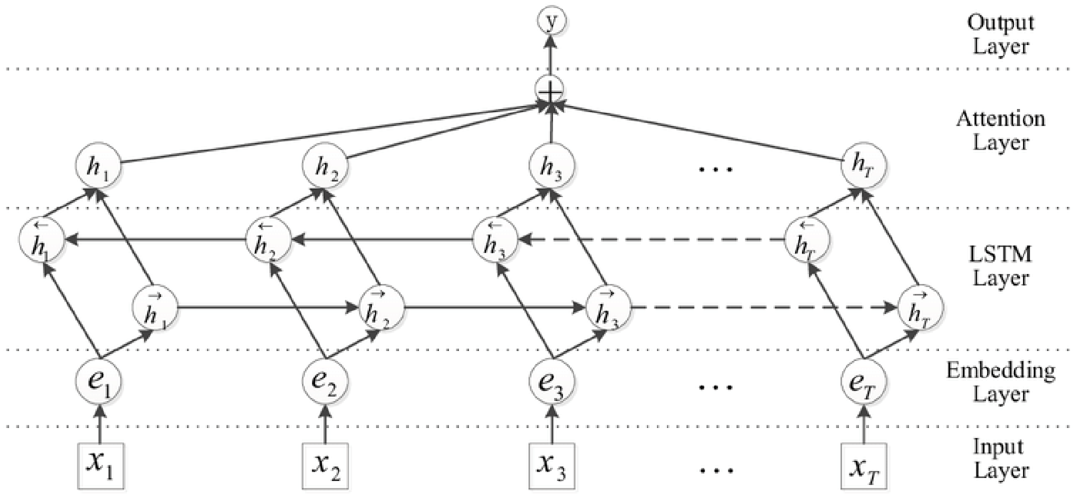

y
Output Layer
Attention Layer
LSTM Layer
Embedding Layer
Input Layer
h_1 h_2 h_3 ... h_T
$\overleftarrow{h}_1$ $\overleftarrow{h}_2$ $\overleftarrow{h}_3$ $\overleftarrow{h}_T$
$\overrightarrow{h}_1$ $\overrightarrow{h}_2$ $\overrightarrow{h}_3$ $\overrightarrow{h}_T$
e_1 e_2 e_3 ... e_T
x_1 x_2 x_3 ... x_T

Figure 4.5: Example of a bi-directional LSTM with an attention mechanism (image taken from [90]).

4.6.3 Gated Recurrent Unit

Recently another gated RNN called the *gated recurrent unit* (GRU) has been gaining in popularity among researchers. Proposed by Cho et al., the GRU has a simpler design by only containing two gates: the update gate and the reset gate [16]. In short, the update gate z combines the forget and input gates of the LSTM, determining what should be kept from the previous memory state. The reset gate r determines how to combine the new input into the cell's hidden state. The equations for these are comparable to the gates in the LSTM:

$$z_t = \sigma_s(W_z x_t + U_z h_{t-1} + b_z), \tag{4.35}$$
$$r_t = \sigma_s(W_r x_t + U_r h_{t-1} + b_r). \tag{4.36}$$

The main difference can be found in how the update and reset gates are used in order to update the hidden state:

$$\tilde{h}_t = \sigma_h(W_h x_t + U_h(r_t \circ h_{t-1}) + b_h), \tag{4.37}$$
$$h_t = (1 - z_t) \circ h_{t-1} + z_t \circ \tilde{h}_t. \tag{4.38}$$

As shown by Equations 4.37 and 4.38, contrary to the LSTM the GRU does not maintain a separate cell state. Instead, only the hidden state is passed on in the sequence. Thanks to its simpler design containing fewer parameters the GRU is faster to train than the LSTM, and has in certain tasks shown comparable or even better performance [18, 43]. However, there is no clear consensus on this topic yet, as others have also argued that the GRU's simplification can negatively impact the performance. This resulted in the LSTM consistently outperforming the GRU in some tasks [40, 14]. It will therefore be interesting to see how the GRU and LSTM compare in practice for our problem.

4.7 Attention Networks

Recently the idea of using attention in neural networks has increased in popularity. The main idea behind attention in NLP tasks is that when an input text sequence is used, not all tokens in the sequence are equally relevant to finding the answer. For example, in order to classify a company Web page that states *"We are a construction company based in Florida"*, the word *"construction"* is much more relevant to make a correct classification than the word *"Florida"*. An overview of an attention mechanism used in combination with a bi-directional LSTM is shown in Figure 4.5. Contrary to the bi-directional RNN classification architecture shown in Figure 4.4, the outputs from all recurrent units in the sequence are used. The attention layer looks at all states, and determines which parts of the sequence are most relevant and should therefore be attended more strongly in the classification.

This attention layer is defined as follows. Let O be a vector of output states $[o_1, o_2, ..., o_N]$ from an RNN where N is the length of the sequence. The attention layer is then implemented as described in [90]:

$$M = \tanh(O), \tag{4.39}$$

$$\alpha = \text{softmax}(w^T M), \tag{4.40}$$

where w^T is a transposed weight vector. This yields the activation vector α, whose activation values can be used to calculate a weighted sum of the output vectors:

$$r = O\alpha^T, \tag{4.41}$$

$$h^* = \tanh(r). \tag{4.42}$$

Finally, the classification for a text sequence S is made using a dense softmax layer like with other approaches, using the weighted output vectors:

$$\hat{y} = \text{softmax}(W^{(S)}h^* + b^S). \tag{4.43}$$

As the weights in the attention layer are trained, it can learn which parts of a text sequence it should pay more attention to. Zhou et al. showed that a simple model as shown in Figure 4.5 was able to classify texts with similar or higher accuracy than more complex models [90].

4.7.1 Hierarchical Attention Network

Yang et al. introduced the *Hierarchical Attention Network* (HAN) for document classification [85]. The HAN uses a similar attention mechanism to the one described in the previous section, only it is applied to two levels. As shown in Figure 4.6, they inserted attention layers for both individual words and sentences. This hierarchical approach allows it to not only put a weight on the importance of individual words, but also on that of sentences. This also made it possible to see which inputs a classification is based on by extracting the activation vectors for a sample.

First, words are transformed to word embedding vectors. These vectors are then encoded using a bi-directional RNN, which is a GRU in the original implementation. For each sentence, the output states from the GRUs are used as inputs for the attention layer, which is implemented similarly to Equations 4.39-4.41. These weighted vectors are then used as input for the second layer of bi-directional GRUs, which acts as the sentence encoder. To increase the weight of sentences that help more towards classifying the document another attention layer is added on top of the sentence encoder. The output of this second attention layer is used to classify the document in a dense softmax layer, like in Equation 4.43.

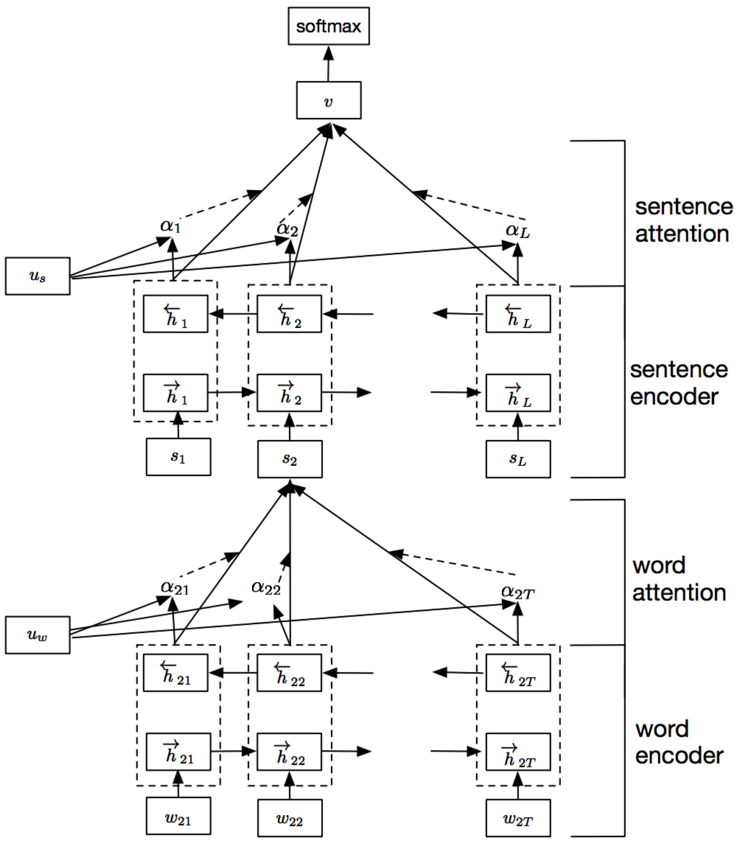


Figure 4.6: Overview of the architecture of the HAN, using attention layers at both the word and sentence level (image taken from [85]).

Chapter 5

Experimental Setup

To answer the research questions posed in the introduction, we benchmarked several of the methods mentioned in the previous chapters against real-world datasets. In this chapter we show how these experiments were set up, what data were used and how the models were configured. First we explain how the Web pages that were used in the experiments were acquired and preprocessed. We then discuss the datasets used in both the keyphrase extraction and Web page classification experiments. Finally, the methods and word embeddings that were used for the keyphrase extraction and classification are discussed.

5.1 Text Extraction & Preprocessing

In order to use the data from a Web page, the page first has to be parsed to extract all relevant text. This parsing of raw HTML-code was done using the BeautifulSoup[1] Python package. A potential problem in extracting text from HTML is that some elements are used to stylize words in sentences, or even parts of words. As an example, consider the following snippet of HTML code:

```
<div>
    <b>J</b>oe's <b>R</b>estaurant
</div>
```

In this case the text 'Joe's Restaurant' should be extracted, but when all tags are considered to be sentence separators this results in this small text fragment being separated into four words. This happens because the bold tag <*b*> is not used as a structural, but as a styling tag. Therefore all tags that were considered style tags were ignored, moving their content directly into the parent node. Table B.1 in Appendix B shows all HTML-tags that were marked style tags. This table also shows which tags were considered 'invisible', meaning that their contents are not displayed when the page is rendered. For example the script tag is generally filled with code that is executed when the page is rendered, and does not contain any relevant text. All content within these tags was therefore not considered. Two exceptions to this are the title and the description, which are often in the head and meta tags, as these do tend to contain text relevant to the page and company.

After all text was extracted, the pages that were used for the classification task were further processed by removing all non-alphanumeric characters. We also found that in some cases the homepage of a company contained very little text. An example of this is that some websites start with a landing page that acts as a welcome screen, often containing only an image and a link that says *"click here to access our site"*. Additionally, in some cases the domain name of a company was expired. In this case the domain could still be reached, however the content had been replaced by a short message stating the domain has expired. As there is not enough relevant content on these types of Web pages for correct classification, all documents that contained

[1]https://www.crummy.com/software/BeautifulSoup/

fewer than 25 words after the preprocessing step were excluded. We also experimented with stemming all words using a Porter stemmer before classification, a common preprocessing step in NLP. Interestingly, preliminary tests showed lower classification scores for classifiers trained on the stemmed data, so this step was omitted from the process. The maximum amount of words was limited to 1,000 to trim down excessively large Web pages. We also attempted to decrease this number to improve the efficiency of the classifiers, however in preliminary tests we found that using less than 1,000 words significantly decreased the accuracy.

5.2 Datasets

5.2.1 Keyphrase Extraction

In order to evaluate the quality of keyphrase extraction methods there are several annotated datasets available that are frequently used as benchmarks. Examples of these are the Inspec dataset consisting of scientific abstracts [38], the keyphrase extraction part of the SemEval dataset on scientific papers [46] , and the set of news articles from [56]. For Web pages however, as far as we know, no such benchmark exists. Given this lack of a gold-standard dataset, we created our own. In order to select a set of Web pages that was representative of the content on the Web we used Dataprovider.com's database of domains, which contains over 250 million indexed hostnames. From this database, 105 hostnames were randomly selected, using some filters to improve the quality of the data. These filters constrained the data to Web pages where English was detected as the main language, as our tokenizer and embedding model were trained on English texts. We furthermore included the constraints that the homepage still had to be online, was not a placeholder or parked domain, and contained at least some text. Three annotators were asked to annotate the complete set by going to each Web page, and to write down the keyphrases they thought best described the Web page. They were given three main guidelines:

- Keyphrases must be present on the homepage (text in title or meta description is also allowed).
- Keyphrases consist of at least 1 and at most 5 words.
- Choose at least 3 and at most 8 keyphrases per page.

Two of the websites were further rejected for not containing enough content, resulting in a final dataset of 103 annotated Web pages. Similar to [32] and [5], the union of the three sets of keyphrases for each Web page was used for the evaluation. The final combined dataset contained an average of 10.02 keyphrases per document, while each annotator assigned on average 5.19 keywords per document. In order to detect wrong or misspelled annotations, we removed all annotations that were not in the extracted text from the page. The text extraction and preprocessing was performed only once, after which all parsed content was stored in page snapshots. This ensured that all methods were evaluated using identical Web pages as input. To avoid copyright infringement we can not share the exact snapshots of the Web pages used for the evaluation, however we published[2] the set of annotations such that other researchers can use this for their experiments, and can hopefully extend it.

5.2.2 Classification

To evaluate the Web page classifiers, we trained the methods to predict two levels of *Standard Industrial Classification* (SIC) codes. The SIC categorization is a standardized indexation developed to assign companies to predetermined categories. The categorization is hierarchical,

[2]https://gitlab.com/Tim-Haarman/WebEmbedRank

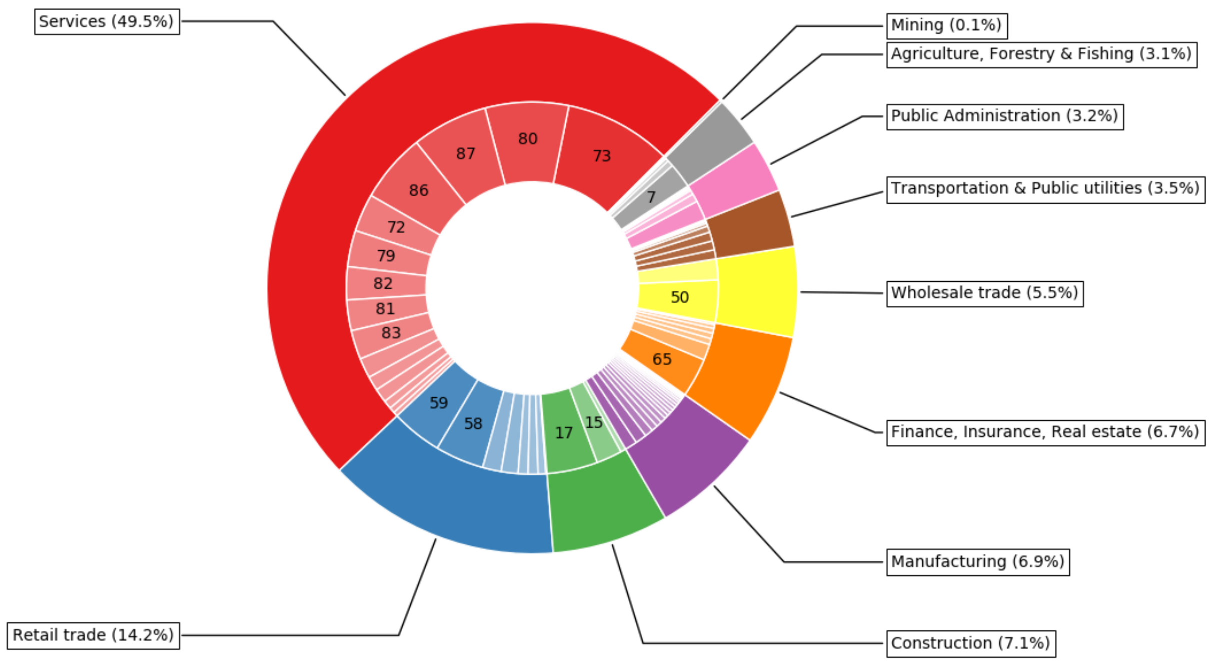


Figure 5.1: Class distribution of first and second level SIC-codes. The outer layer show the 10 divisions, the inner layer the 84 second layer two-digit codes.

meaning that there are several levels of categories, with each successive category more specific than the last. In our tests we will be training on the two highest levels. The first level, also known as the Division, contained 10 main categories. The second level was significantly larger, containing 84 categories, which we will refer to as the SIC category. The labeled data for the supervised classification of SIC-categories came from *Dunn & Bradstreet*, a company that maintains a business directory that contains the SIC categories businesses were registered with. From these data all companies that had a registered domain name were extracted, along with their annotated SIC code. This yielded an annotated dataset of 1,237,813 company Web pages.

An important characteristic of the data was that the class distributions were strongly imbalanced. Both the first level (the Division) as well as the second level distributions were heavily skewed. Figure 5.1 provides a depiction of the class distributions, showing all distributions of the 10 main divisions and the largest second-level codes. This for example shows that close to half of the data was in the *'Services'* category, whereas only approximately 0.1% of all companies were in the *'Mining'* category. A full list of all first-level divisions and second-level SIC codes are provided in Table A.1 in Appendix A for reference.

5.3 Keyphrase Extraction

In order to evaluate the performance of our keyphrase extraction method, we compared it to several baselines and state-of-the-art extraction methods. Similar to earlier research [38, 59, 24, 10], the methods were evaluated based on their precision, recall, and F_1 score with respect to the Web page annotations. In the matching of the keyphrases we only accepted exact matches of the complete phrase. We will briefly describe the evaluated methods.

The first method was *TFIDF* [75], a frequently used and robust baseline, implemented as explained in Section 3.1. In order to determine the document frequencies we used the large company Web corpus we created for the SIC classification, as described in the previous section. The top keyphrases were selected by taking the phrases with the highest TFIDF scores. The *location based* method was a simple heuristic baseline that was based on the aforementioned theory that certain HTML elements have varying levels of relevance. It applied a part-of-speech (POS) tagger to all texts, and selected all the longest chains of zero or more adjectives followed by one or more nouns as candidates. The candidates were ranked based on the HTML element they appeared in. This hierarchical ranking method ranked candidates that occurred in the hostname the highest, followed by the title, headings, description, and all remaining elements respectively. If multiple candidates were present in an element they were ranked in the order of appearance in the original HTML code.

We further compared the WebEmbedRank and WebEmbedRank++ methods from Section 3.4 to four state-of-the-art graph based models[3] as described in Section 3.3, being TextRank [59], TopicRank [11], PositionRank [24] and MultipartiteRank [10]. Lastly we also compared it to the original implementation of EmbedRank[4] [5], using both the standard version (EmbedRank) and the version with diversity mechanism (EmbedRank++). For all methods we used the versions and parameters as suggested by the authors. The tokenization and POS-tagging was done using the Stanford CoreNLP toolkit [55] for all methods.

5.4 Web Page Classification

5.4.1 Word Embeddings

Training custom word embeddings from scratch can potentially create a better vector space for a new domain, especially if the text in the domain diverges much from the domain the pre-trained embeddings were trained on. However, as our data consisted of Web pages across a large variety of domains that were likely to mostly consist of generic terms, we expected that the size of the dataset the pre-trained embeddings were trained on outweighed the benefits of training new embeddings from scratch on a custom dataset. We therefore chose to use and compare various pre-trained word embeddings.

For all embedding methods we chose to use 300-dimensional embeddings, as this is one of the most commonly used sizes, in which most pre-trained models are available [86]. The Word2vec model was obtained from the official Word2vec project page[5], and was pretrained on a Google News dataset containing more than 150 billion tokens. The GloVe embeddings were pretrained on a combination of a Wikipedia archive and the common crawl dataset, as described in [67]. The files were the same as in the paper, and were obtained from GloVe's official Web page[6]. Finally, the fastText embeddings were also obtained from the official website[7]. These embeddings were trained on a combination of Wikipedia data, Stanford's web-based UMBC corpus, and a news article dataset from statm.org [60].

We also implemented an ELMo embedding layer, however found it to be too memory intensive for our current systems. To get a fair comparison we used the first 1,000 words of the text like with the other methods, but even when using stochastic gradient descent training one sample at a time, it ran out of the 4GB of VRAM available on our GTX 970. We also tried running it on a CPU, using the 32GB of normal RAM. This was possible, however the training was estimated to take over a month per epoch, rendering this approach unfeasible with our current setup.

[3]TextRank, TopicRank, PositionRank and MultipartiteRank were implemented using the python keyphrase extraction (PKE) toolkit [9]

[4]EmbedRank and EmbedRank++ were tested using the original implementation of the authors: `https://github.com/swisscom/ai-research-keyphrase-extraction`

[5]`https://code.google.com/archive/p/word2vec/`

[6]`https://nlp.stanford.edu/projects/glove/`

[7]`https://fasttext.cc/docs/en/english-vectors.html`

Table 5.1: Results of grid-search on bag-of-words classifiers. F_1 indicates the mean micro-F_1 measure, s.d. shows the standard deviation on the test score over the 5 folds.

	Classifier	**F_1**	**S.d.**	**Best model configuration**
Division	LR	0.745	0.004	Penalty (C) = 1.5
(Level 1)	RF	0.677	0.003	Gini impurity, num. trees = 100, max depth = 500, min. samples per split = 2
	SVM	0.748	0.003	Squared hinge loss function, penalty (C) = 0.15
	MLP	0.748	0.003	1 hidden layer, layer size 50, ReLU activation
SIC	LR	0.601	0.003	Penalty (C) = 2
(Level 2)	RF	0.542	0.004	Gini impurity, num. trees = 200, max depth = 250, min. samples per split = 2
	SVM	0.609	0.002	Squared hinge loss function, penalty (C) = 0.15
	MLP	0.601	0.003	1 hidden layer, layer size 100, tanh activation

5.4.2 Baselines

All tests with the baselines were performed using a bag-of-words (BoW) representation of the complete set of parsed text from the Web page. We furthermore applied a TFIDF transformation on the input, increasing the values of more discriminative words (as explained in Section 3.1). Preliminary tests showed that this transformation consistently increased the performance of the baselines. To avoid overfitting and somewhat limit the dimensionality of the input data, only the 20,000 most common words with respect to the term frequency across the corpus were considered. For all BoW based classifiers the hyperparameters were determined using exhaustive grid searches. As the different amount of classes in the first and second levels of the SIC codes may influence the optimal choice of parameters, the grid searches were performed separately for both levels. This process can be very computationally demanding depending on the amount of parameters considered, as it requires retraining a classifier multiple times from scratch. Therefore, for all hyperparameter optimization tests a seeded random sample of 100,000 data points was used. All grid searches were performed with 5-fold cross-validation, using the micro-F_1 score as performance measure to account for the class imbalance. For all classifiers a training split of 80% was used. The remaining 20% was used as test set for all classifiers except the multilayer perceptron, which used 10% as a validation set and the remaining 10% as a test set. When the highest score was achieved by multiple sets of parameters, the simplest model was chosen.

For the random forest the Gini impurity measure always outperformed entropy. Additionally, because the Gini measure did not require logarithms to be computed (see section 4.2), entropy also was approximately 9.5% slower to train on average over all folds compared to Gini. The MLP on the division level classification showed the same average score of 0.748 for both the sigmoid, tanh and ReLU activation functions. Of these three the ReLU activation function showed the fastest training times and was therefore chosen. For the second level codes the tanh showed a slightly better score. The reason the ReLU function was not better despite its recent popularity is likely because the best model consisted of a single hidden layer. As explained in section 4.4.3, the main reason ReLU is currently the preferred activation function is that it works well with deeper neural networks. With just a single hidden layer, these benefits are not as relevant.

5.4.3 Neural Networks

For all word-level neural networks the text was first tokenized, and converted to word embeddings to serve as input. For the CNNs the input was padded to a length of 1,000 where necessary. The standard CNN was implemented to act as a baseline for the convolution based methods, which was empirically tuned for this task. It used two convolutional layers, each consisting of 128 dimensions with a filter size of 5. Both of these were followed by a max-pooling layer of size 5 and 3 respectively, to which dropout was applied with a 30% chance. These layers were followed by a dense layer of size 128, followed by the variably sized softmax output layer. Between the last two layers dropout was applied with a 50% chance.

The C-CNN was implemented as described in [47]. We used convolutional filters of size 3, 4, and 5, which each constructed 100 feature maps, and applied a dropout with probability 0.5 on the penultimate layer. The only change from the model proposed by Kim was that we used a larger mini-batch size of 256 to speed up the training. For the Character-CNN we used the small network as proposed by Zhang, Zhao, and LeCun [88], as the larger model did not seem to improve the results. We further used the same dictionary of 70 characters without case sensitivity, as the authors showed that this also did not improve the performance. Zhang, Zhao, and LeCun further suggested that taking only the first 1,014 characters is sufficient, which we also mimicked in our classifier. However, this is often only a fraction of the text on a Web page and much fewer than other proposed architectures utilize. We therefore trained another C-CNN on the first 2,028 characters, doubling the suggested amount of input characters. This could indicate whether the performance of the suggested model may have been held back by the limited amount of input it received.

We also wanted to see how well recurrent neural networks could classify the Web pages. As the standard RNN is in practice never used anymore due to its inability to maintain long term dependencies, we only looked at the LSTM and GRU units. We performed a grid-search to find the best number of hidden units to use, and found one layer of 300 units to be optimal while still being trainable in a reasonable time. For all RNNs we used bi-directional layers, therefore resulting in a total of 600 units per layer. To see if the attention module could improve the results, we also trained a bidirectional GRU of 300 units with an additional attention layer as described in [90]. For the HAN a setup was used that was comparable to the one proposed by Yang et al. [85], using the attention mechanism at both the word and the sentence level. For consistency we used 300-dimensional word embeddings, and we increased the dimensionality of the GRU encoder from 50 to 100 as we found that this increased the performance. As the attention layers require a fixed size input vector, the size of the input text had to be limited and padded similar to the convolutional neural networks. As the HAN contained two attention layers, this was required at both the sentence and document level. The size of the sentences was set to 50 tokens, with 100 sentences per document, both padded where necessary.

Chapter 6

Results

In this chapter the results from the experiments described in the previous chapter are presented. First, we discuss the evaluation of our novel WebEmbedRank method for keyphrase extraction. We then show the results for the Web page classification, and present the time each method required to classify new samples.

6.1 Keyphrase Extraction

To compare our WebEmbedRank method to various state-of-the-art approaches, we evaluated the extracted keyphrases using a manually annotated dataset. The resulting precision, recall and micro-F_1 scores for all methods as evaluated on this gold-standard set are shown in Table 6.1. As all keyphrase extraction methods required a predetermined number of keyphrases to extract, the results for both the top-5 and top-10 keyphrases are shown. As the amount of annotated keyphrases varied between Web pages in the test set, achieving an average precision or recall of 100% was not possible.

The results show that WebEmbedRank outperformed all other methods, achieving the highest precision, recall and F_1 score for both the top-5 as well as the top-10 ranked keyphrases. Paired t-tests showed significant improvements of WebEmbedRank over TFIDF, TextRank, TopicRank, PositionRank, MultipartiteRank, EmbedRank and EmbedRank++ for $N = 5$, and over all other tested methods at $N = 10$, with $p < 0.001$. Surprisingly, the simple location based heuristic got the second highest scores. This indicates the importance of considering the hypertext markup in the extraction process. The TFIDF method also scores comparably or even better than some of the state-of-the art methods.

The low scores of the EmbedRank and EmbedRank++ methods are surprising, as the authors have shown that it can achieve state-of-the-art results on similar tasks with datasets of articles and abstracts [5]. In this task they score the worst of all methods, with F_1 scores of only 8.24 and 7.34 on the first five keyphrases. For the top-10 keyphrases the scores are slightly better relative to the other methods, but still low with 16.22 and 10.09 respectively. By inspecting the extracted keyphrases we found that the low scores of many extraction methods may have been due to a tendency to generate long keyphrases. Embedrank and EmbedRank++ also have this tendency, as explained in Section 3.5. The phrases selected by these methods contained on average 2.43 and 2.13 words respectively, compared to an average 1.51 words for WebEmbedRank and 1.61 words for the annotated phrases in the gold-standard evaluation set. We observed the same problem with PositionRank, which extracted phrases averaging 2.36 words.

We furthermore also see that the scores of the EmbedRank and WebEmbedRank versions with diversity do not perform better than the standard versions. This concurs with the results of Bennani-Smires et al. [5]. For the top-5 keyphrases the scores are only slightly lower, and the difference is not significant. When the top-10 keyphrases are considered the difference becomes much greater, going from 10.09 to 16.22 and from 25.82 to 34.81 for EmbedRank and WebEmbedRank respectively.

Table 6.1: Scores of WebEmbedRank compared to various other state-of-the-art keyphrase extraction methods on the Web page dataset. **P** denotes the precision, **R** the recall and **F$_1$** the micro-F$_1$ score. The highest scores are marked in bold.

	N=5			*N=10*		
Method	**P**	**R**	**F$_1$**	**P**	**R**	**F$_1$**
TFIDF	30.49	16.20	20.78	20.97	21.81	21.00
Location	40.00	21.11	27.09	29.42	30.13	29.29
TextRank	15.73	8.37	10.74	14.94	15.54	14.98
TopicRank	29.51	15.47	19.92	22.04	22.86	22.01
PositionRank	24.85	13.43	17.14	21.36	22.61	21.57
MultipartiteRank	32.82	17.24	22.22	24.47	25.47	24.50
EmbedRank	11.84	6.53	8.24	16.02	17.06	16.22
EmbedRank++ ($\lambda = 0.5$)	11.07	5.63	7.34	10.10	10.54	10.09
WebEmbedRank	**43.69**	**23.37**	**29.94**	**34.56**	**36.41**	**34.81**
WebEmbedRank++ ($\lambda = 0.5$)	41.36	22.10	28.27	25.83	26.86	25.82

6.2 Web Page Classification

For the Web page classification we trained multiclass classifiers on the two levels of SIC codes. As mentioned in the previous chapter, the first level that is referred to as the Division contained 10 classes, whereas the second level that is referred to as the SIC level consisted of 84 classes. As the dataset was highly imbalanced, the micro-F$_1$ score was used to evaluate the performance of the models. The results for both datasets can be found in Table 6.2[1].

When the scores are compared at the higher level of model architectures, it shows a clear winner in the methods that are based on recurrent neural networks. For both the 10 and 84 class problems the highest scores are achieved by the GRU with the attention layer, with respective F$_1$ measures of 77.9 and 64.1 for the GLoVe embeddings. However, the standard GRU and LSTM are right behind it, with only a marginal difference in F$_1$ score of 0.1. It also shows that despite its simpler architecture, the GRU performed on a similar level as the LSTM, while even slightly outperforming it for the Division level. For the SIC level the results are reasonably consistent between the LSTM and GRU.

The low scores for the HAN are furthermore surprising. Whereas the GRU with a simple attention layer scores similar to the GRU and LSTM, the more specific HAN fails to keep up. The results show that it scored similar to the baselines for the Division level, and more than a full percent lower on the SIC category. However, the HAN was not the only method that had difficulties in surpassing the scores of the baselines. This was especially the case for the SIC category, where only the RNN-based models outperformed the linear SVM. The Char-CNN also underperformed compared to the baselines, achieving some of the lowest scores all around. Doubling the amount of input characters largely increased the score from 73.5 to 75.1 for the Division, and from 56.6 to 58.4 for the SIC level. This is an indication that only considering the first 1014 characters as suggested by the authors was a limiting factor for this task. Due to time restraints we did not investigate whether further increasing the number of input characters can result in competitive scores, but this may be interesting for further research.

We also see that the baseline CNN scores are just above the best BoW baselines, but for the SIC level it scored comparatively lower by a relatively large margin. We furthermore see

[1]Even with aggressive pruning of the trees and limiting the number of estimators, the Random Forest required more than the 32GB of RAM available for the SIC level classification. Because of this we were unable to collect results for this dataset.

Table 6.2: Results of classification for different embedding and classification methods. The reported scores are the micro-F_1 measures for the first level (division) and second level (SIC) category classification tasks, with the best results marked in bold.

	Classifier	**Embedding**	**Division**	**SIC**
Baselines	Logistic Regression	BoW	76.4	63.1
	Random Forest	BoW	72.5	-
	Linear SVM	BoW	76.5	63.4
	MLP	BoW	76.9	62.9
CNNs	CNN	Word2vec	76.9	62.0
		GloVe	77.0	61.8
		FastText	77.0	62.2
	C-CNN	Word2vec	77.2	63.0
		GloVe	77.1	62.8
		FastText	77.1	62.8
	Char-CNN (1014 ch.)	One-hot	73.5	56.6
	Char-CNN (2028 ch.)	One-hot	75.1	58.4
RNNs	LSTM	Word2vec	77.6	64.0
		GloVe	77.6	63.9
		FastText	77.7	64.0
	GRU	Word2vec	77.7	64.0
		GloVe	77.8	63.7
		FastText	77.8	64.0
Attention	Att-BGRU	Word2vec	77.8	63.9
		GloVe	**77.9**	**64.1**
		FastText	77.7	64.0
	HAN	Word2vec	76.5	61.7
		GloVe	76.4	61.7
		FastText	76.4	61.7

that the multiple filters introduced by the C-CNN did improve on the baseline CNN network, even though it only contained one convolutional layer. Especially on the SIC level it performed better than the CNN, however it still did not beat the best BoW baselines. Overall the convolutional neural networks struggled more with the high number of classes than the BoW classifiers, while performing better on the Division task with 10 classes.

Finally, the different embedding methods did not seem to have a great effect on the final scores. The difference in scores between the methods only slightly varied, and not consistently in favor of one method. Across all methods the three methods score approximately the same on average.

6.2.1 Timing

To find the empirical efficiency of each classification method, we tracked the time each method required to classify new samples. This was done by measuring the time it took each method to classify the complete test set. In order to make the results more reliable, the average time of three predictions was used for each method. This number was scaled to find the mean time it took to classify 10,000 samples, the results of which can be found in Table 6.3. As train and test sets were the same for each method, all timings were based on the same samples. All baselines,

Table 6.3: Mean prediction times in seconds for the various classifiers per 10,000 samples.

		Time (s)	
	Classifier	**Division**	**SIC**
Baselines	Logistic Regression	0.020	0.133
	Random Forest	0.371	-
	Linear SVM	0.019	0.115
	MLP	0.082	0.285
CNNs	CNN	2.24	2.36
	C-CNN	5.06	5.09
	Char-CNN (1014 ch.)	14.13	14.24
	Char-CNN (2028 ch.)	17.96	17.98
RNNs	LSTM	324.73	336.39
	GRU	223.03	255.94
Attention	Att-BGRU	229.59	229.71
	HAN	50.96	51.16

including the MLP, were trained and tested on an i7-6770 CPU with 4 cores and 8 threads. All other methods were trained and tested on a Nvidia GTX 970 GPU with 4GB of VRAM.

The timings show that the baselines could very quickly classify a large number of samples. The linear SVM was the fastest, requiring only 0.019 seconds to assign Division categories to 10,000 Web pages. The simple MLP also only took 0.082 seconds to do this, even though it was tested on the CPU. We do see a large relative increase going from the Division to the SIC classification, but the times are still fast enough to be used on a large scale.

Even though the non-baselines were able to use the computational power of a dedicated graphics card, the timing clearly shows the increased complexity of the deep neural networks. Whereas most baselines were able to classify 10,000 new samples in a fraction of second, the recurrent neural networks take up to 324.73 seconds to do so. This shows that the increased performance as displayed in Table 6.2 comes at a sizable cost in terms of computational efficiency, even compared to the convolutional neural networks.

Chapter 7

Discussion & Conclusion

In this chapter we discuss the findings presented in the previous chapter, and conclude the thesis. First the keyphrase extraction is discussed, followed by the Web page classification. This includes the findings from the results, and suggestions for follow-up research to address items that could not be covered in this thesis. We finally wrap up the thesis with a concise conclusion.

7.1 Keyphrase Extraction

In the introduction we raised the question how the most relevant keyphrases can be extracted from a Web page. As the amount of research on keyphrase extraction that has a focus on Web pages is limited, we attempted to apply techniques that have been shown to work well in other domains to our current task. However, the results of the keyphrase extraction experiment show that most state-of-the-art unsupervised keyphrase extraction methods transfer poorly to Web documents. Multiple methods had a tendency to extract very long phrases, which may explain the lower scores. This might have been due to the different nature of the data, as most keyphrase extraction systems are evaluated on datasets of scientific abstracts and articles. These documents often comprise multiple complex topics, which may require longer keyphrases to accurately represent the document compared to Web pages. To check this hypothesis we analyzed the assigned keyphrases in the training part of the Inspec [38] and SemEval 2010 [46] datasets, which are some of the most frequently used datasets to evaluate unsupervised keyphrase extraction methods [32]. We found that they respectively contained on average 2.33 and 2.16 words per phrase, which is substantially more than the average of 1.61 in our annotated Web page dataset. This disparity can potentially explain why some methods are more geared towards extracting longer phrases, and hence why their results are poor when transferred to the present domain and evaluation set.

Because WebEmbedRank incorporates a small penalty for long phrases, it did not suffer from this problem. Longer phrases could still get a high rank if they got a high similarity score, or if they appeared in important positions, but a preference was given to shorter sentences that could convey a similar meaning. This resulted in keyphrases with an average length of 1.51 words, similar to the annotated set. The additional weighting of keyphrases in important locations furthermore proved to be effective, resulting in the highest scores for both the top-5 and top-10 keyphrases. The importance of considering HTML elements in the process of extracting keyphrases from Web pages is further accentuated by the high score of the location-based method. This method ranked the phrases purely based on their location in the HTML, and achieved the best results behind WebEmbedRank.

When only considering the state-of-the-art models, MultipartiteRank achieved the highest F_1 scores. A part of its success can be attributed to the positional component in this method. The parsed Web page document that was used as input for each method was constructed hierarchically, starting with the more important elements (as defined in Section 3.5). This meant that the positional element in MultipartiteRank effectively assigned phrases from important elements a higher weight, as these were placed at the beginning of the document. PositionRank also implemented a similar positional weighting, but the scores of this method also suffered from extracting phrases that were too long. As PositionRank constructs phrases by summing the scores of the individual words, a strong bias towards long phrases is created, even when some of the words do not contribute much to the overall score. This is a known problem with this approach to constructing phrases, and was earlier criticized by Bougouin, Boudin, and Daille [11]. In the original paper where PositionRank is proposed this problem is largely avoided by limiting the keyphrase extraction to at most tri-grams [24], however it can be argued that such a hard cut-off is undesirable. Unlike PositionRank, MultipartiteRank did not suffer from generating too lengthy keyphrases. The extracted phrases contained on average approximately 1.61 phrases, almost identical to the annotated set.

Similar to the results reported in [5], the EmbedRank and WebEmbedRank versions with diversity mechanism both scored lower than their counterparts without diversity. An interesting finding is that while the scores with and without diversity are quite close for the top 5 phrases, they are much higher when the diversity method is enabled for the top 10 keyphrases. This may be explained by the lack of enough diverse keyphrases on a Web page. Often the aim or topic of a Web page is quite specific. This is also shown by the fact that the annotators assigned 5.2 keyphrases on average per Web page, even though they could provide 8. Forcing diversity during the generation of 10 keyphrases may result in irrelevant phrases being extracted when there are only a few true main topics. However, Bennani-Smires et al. also showed a pilot experiment which suggested that users preferred the keyphrases with diversity, even though it achieved lower scores on their evaluation set as well [5]. To see whether this is also the case for WebEmbedRank and WebEmbedRank++ it may be interesting to follow up on the current experiments with a similar user study.

In this thesis we focused the weighting on what we perceived to be the most important elements, being the hostname, title, headings and description. Earlier work has shown some indications that other elements may also have positive or negative effects on the relevance of their contents, such as text style elements (bold, italic, etc.) or anchor text [21, 17]. We could therefore extend the current system to include weights for more elements. Additionally we may experiment with different weights for the different sizes of headings (h1 to h6), instead of having one fixed weight for all sizes.

We furthermore exclusively considered English Web pages in the current work. However, given the unsupervised nature of WebEmbedRank, it is expected that it can be generalized to other languages relatively easily. The word embeddings can be trained in the same way for different languages, as long as there is some large set of written text available in that language. Given that the fastText method already provides pre-trained embedding models for 157 languages [27], this is not expected to be a problem. Different languages will also require an automatic POS-tagger trained for that language. This may be problematic for smaller languages, but the Stanford CoreNLP POS-tagger we used in the experiments is available in most major languages [55]. Whether the same rules about what constitutes a good keyphrase also apply to different languages remains to be answered, which we will leave for future work.

7.2 Classification

To find the classification method that could most accurately classify Web page categories we compared several popular text classification models. The effectiveness of these methods were evaluated by training them to predict two levels of industrial categorization codes. In terms of the micro-F_1 measure, the LSTM, GRU and Att-BGRU achieved the highest scores, with the Att-BGRU scoring slightly higher for one of the three embedding methods. As this difference was so small, it remains questionable whether the added attention layer significantly improved the GRU. Overall, these RNNs clearly outperformed the baselines and CNNs.

Interestingly, the character level convolutional neural network scored quite poorly, performing worse than most baselines. A likely cause for this can be found in the amount of input data that was used. The model we used was based on the parameters as proposed by Zhang, Zhao, and LeCun in their paper [88]. In this model only the first 1,014 characters of a document were used, likely in order to keep the resources required to train the model feasible. Considering each character individually can potentially make the model more robust as it can make it less vulnerable to misspellings and out-of-vocabulary words, but it also greatly increases the computational complexity. In our preliminary tests with the bag-of-words classifiers a significant drop in accuracy happened when limiting the document length to 500 words instead of the 1,000 used currently. Reducing the information a model gets to only 1,014 characters may have severely impacted the performance of the model. To confirm that this was a limiting factor, we also tested a second model with the same hyperparameters, apart from that is used the first 2,048 characters. This doubling of the input size increases the accuracy from 73.5 to 75.1 for the division and from 56.6 to 58.4 for the SIC category, indicating that using only the first 1,014 was likely not enough. Additionally, Zhang, Zhao, and LeCun say: *"only until the dataset goes to the scale of several millions do we observe that character-level ConvNets start to do better"*. Although our dataset consisted of over a million documents, perhaps even more data was required to achieve competitive scores.

The HAN also did not perform well compared to the other models. We expect that this can largely be attributed to the inconsistent length and content of both complete documents and the individual sentences. Because the consecutive attention layers require a fixed amount of inputs, all sentences and documents had to be padded to a static length. Choosing this length results in a trade-off. By increasing the length less information is lost as long sequences are not cut short, but this also results in a high number of padding values, which makes the process much more inefficient. When the length of the input data is very inconsistent the effects of this trade-off become more apparent. With Web pages this is a problem, as they often contain many sentences that consist of only one or a few words, like headings, titles, and menu items, yet can also contain full length paragraphs. Yang et al. also explicitly mentioned that they chose documents with similar lengths for their experiments [85], which may be why they were able to get good results with this architecture. Similarly, the CNNs also required inputs of a fixed size, and may therefore also have suffered from this problem.

Furthermore,the relatively high baseline scores for the classification may be due to the nature of the task. For many complex NLP challenges that evaluation measures are based on, getting the right classification can be highly dependent on the context and order of words in a sentence. For example, in a sentiment analysis task there is a large difference between *'I did order the lobster, but it was not good'* and *'I did not order the lobster, but it was good'*. Similarly, in question answering tasks *'Is A larger than B?'* requires a different answer than *'Is B larger than A?'*. Contrary to this, in the current classification task if the word *'restaurant'* and *'food'* appear multiple times on a page it is relatively clear that it belongs in the food services category, regardless of the exact context the word appears in. In this case a bag-of-words approach is more likely to be sufficient.

Nevertheless, the best model had F_1 scores of 77.9 and 64.1 for the two classes, indicating that the tasks were not trivial. This can be attributed to a few things. Firstly, the large amount of classes (10 and 84 for the division and SIC respectively) increased the difficulty and base-rate of making a wrong prediction. Furthermore, and perhaps more importantly, the assigned label was not always exclusively the right one. The dataset contains user-assigned labels, with one main label for each company. The problem with this is that some companies could fit well in multiple categories. For example, a car dealer that both sells and repairs cars could be assigned the label G-55 (Automotive Dealers) or I-75 (Auto Repair, Services & Parking). Both would be

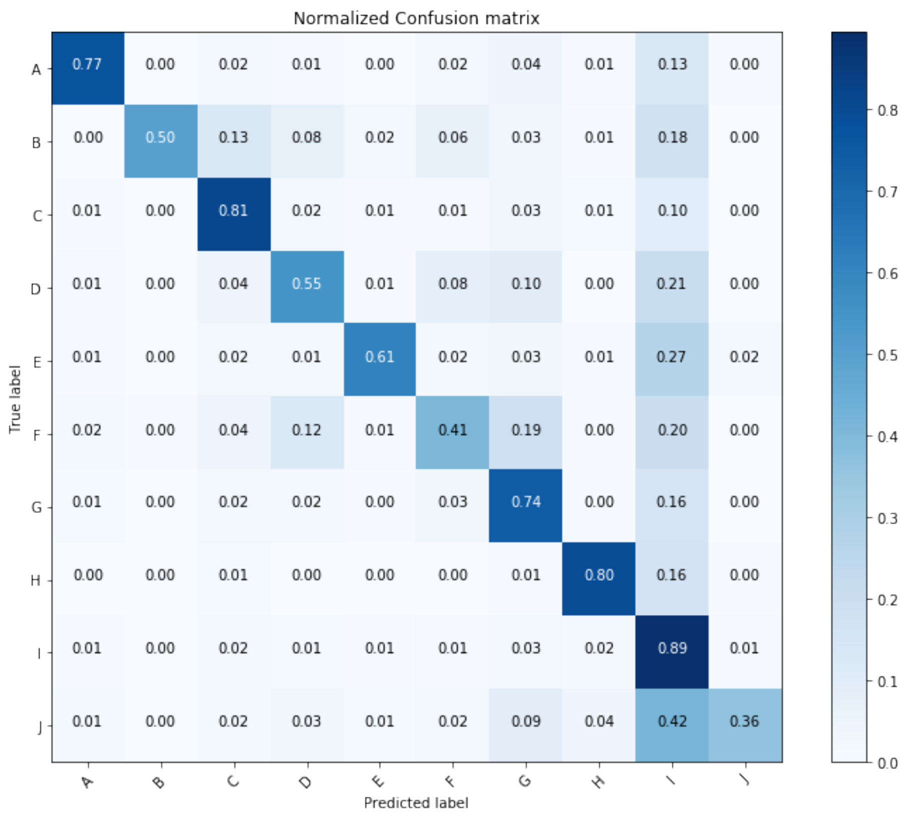


Figure 7.1: Normalized confusion matrix showing the ratio of classifications for each class for the GRU classifier using fastText embeddings.

correct labels, yet they have a different code and even are in different divisions. Additionally, the difference between some labels can be quite subtle, especially in the SIC level. This is for example shown by the fact that on the second level there are separate categories for *Railroad Transportation* (40), *Passenger Transit* (41), *Water Transportation* (44), *Air Transportation* (45), and *Transportation Services* (47).

Figure 7.1 furthermore shows that many samples are wrongly classified as the *I* division, which is the Services category. This is caused by the imbalanced nature of the dataset, as illustrated in Section 5.2.2. Due to its large base rate, the classification is biased towards the *I* class. In practice this results in the model assigning all samples it is not confident about to the *I* class, as this yields the overall highest accuracy. This can be avoided by taking a stratified sample to undersample the majority classes, or by using an oversampling technique for the minority classes like SMOTE [15]. Although this may increase the mean classwise accuracies, this is likely to decrease the overall (micro) accuracy. We did experiment with both under- and oversampling, but opted not to use it as we were more interested in the overall accuracy of the model.

7.2.1 Word Embeddings

We furthermore set out to find the effect of different word embedding techniques on the classification of the Web pages. To find an answer to this question, we applied frequently used pre-trained models of Word2vec, GloVe and fastText, as provided by the authors of each method. The results seem to indicate that the different word embedding methods that were used did not have a large effect on the performance of the classification models. The fastText model scores slightly higher with some classifiers, but this result is not statistically significant. However, as the fastText embeddings achieved good results and are fast to train, we are inclined to prefer this method. As fastText embeddings can theoretically also better deal with a high number of declensions and compound words, they may show better results when retraining for a different language like German. Whether this is also the case in practice remains to be answered.

We furthermore implemented ELMo embeddings [69], yet found that they were too resource intensive to be used effectively for this task. As ELMo works on a character level, it creates separate embeddings for each character in the input. While this has shown to be effective, it is also hugely memory and processing intensive. This practical constraint is often circumvented by evaluating it on NLP tasks with short texts, like question answering or text entailment [69]. Contrary to this, our dataset is both very large with more than a million samples, and has highly varying text lengths. We were able to use ELMo embeddings when limiting all documents to the first 128 words, but the discarding of large amounts of text was more detrimental to the results than the contextual embeddings could make up for, and did not offer a fair comparison to the other methods. For now we have to conclude that while the technique is very promising, it is not practical to use on a large scale for long documents.

For future research it may be interesting to experiment with training embedding methods from scratch. Currently only pre-trained embeddings were used that were mostly trained on data from a different domain than Web content. By completely training the methods on hypertext data, the resulting vector space model may be better constructed for Web page classification.

7.2.2 Classification Efficiency

Finally, we aimed to empirically find the computational efficiency of the tested classification methods to assess their viability in a large scale setting. To this end we measured the time each method required to classify new samples. Although the recurrent neural networks achieved the highest classification scores in terms of F_1 score, they are not intrinsically the go-to choice in a production setting. Comparing the GRU with the MLP showed a slight improvement of less than 1% in terms of accuracy, however the time required to classify a sample increased greatly. For the division data, going from an average of 0.08 to more than 223 seconds per 10,000 samples indicated an increase with a factor of more than 2,720. The linear SVM was even faster, and although its accuracy was slightly lower than the MLP it classified the samples more than 11,738 times faster than the GRU. Additionally, it has to be noted that the baselines ran on a CPU with 8 threads, whereas all other methods used a dedicated GTX 970 GPU. When we only look at the highest scoring RNNs we do see the increased efficiency from the simpler structure of the GRU compared to the LSTM, while achieving similar scores. This concurs with earlier findings in [18, 43].

Unsurprisingly, making predictions is much slower for the logistic regression, linear SVM, and MLP classifiers when classifying the SIC labels compared to the Division labels. As explained in Section 4.1, logistic regression and the linear SVM use the one-vs-all classification scheme for multinomial classification. This means that for each extra class label an extra classifier has to be used. Whereas the Division level has 10 classes, the SIC has 84. Consequently, the classification of the SIC codes roughly takes 8.4 times longer for these models. For the MLP the increase in time is caused mostly because we used twice as many nodes in the hidden layer for the SIC model, as explained in Section 5.4.2. Additionally, the increased amount of classes may have also caused the efficiency to decrease, as it requires the calculation of more weights at the output layer, and the softmax function has to be computed over more outputs. This effect is also present in the more complex neural networks, although the relative increase is not as large as these have many more weights to train that do not depend on the size of the output layer.

In the end, the final decision on which method is most suitable for this task comes down to the scale to which it is applied and the computational resources that are available. To illustrate the potential impact of this increase in complexity, we can extrapolate the result to Dataprovider.com's complete Web page database. In total, this database contains over 281 million indexed Web pages. If we were to classify these on the Division level with an LSTM, it would approximately take more than 100 days to complete on a GTX 970 GPU. However, if we were to accept a slightly lower accuracy and use the linear SVM, the complete database could be classified in less than 9 minutes on one CPU.

7.3 Conclusion

In this thesis a new unsupervised method is proposed to extract keyphrases from Web pages. It combines the semantic information provided by word embedding techniques with domain knowledge about the structure of Web pages to effectively select the most representative phrases from a Web page. We furthermore compared and evaluated several approaches to the classification of Web pages, and tested the effects of three different text embedding methods.

For the keyphrase extraction we proposed WebEmbedRank, an extension to EmbedRank [5] for keyphrase extraction from Web pages. WebEmbedRank uses text embeddings to extract only the phrases that are relevant to the document, and strengthens the ranking with a weighting procedure based on the structural information in the document. To evaluate this novel approach, we annotated keyphrases for 105 randomly sampled Web pages. We then compared the extracted keyphrases of WebEmbedRank and several state-of-the-art methods with these manually annotated keyphrases. This showed that the keyphrases extracted by our new method were significantly closer to the gold-standard set than all other compared methods. It also showed that considering the structural markup that Web pages offer is crucial in determining good keyphrases. By using this structural information where available to strengthen the ranking process, but not solely relying on it, WebEmbedRank was able to effectively deal with the inconsistency in Web documents.

We furthermore looked at different supervised classification methods to categorize Web pages. We found that recurrent neural networks consistently outperformed various baselines and convolutional neural networks. However, this higher accuracy also came with significantly increased prediction times for new samples. Choosing a suitable method for large scale use therefore requires a careful consideration of the required classification accuracy and the available computational resources. We also found that most convolutional approaches did not work as well due to their requirement of fixed sized input. As the content on Web pages is very inconsistent in length, this resulted in either efficiency or efficacy problems. We additionally reviewed the effect of various pre-trained word embeddings on the classification, which showed that overall the choice between pre-trained embedding methods did not have a significant impact on the results. All methods seemed to generalize well, and the small differences may have easily been rectified by fine-tuning the embeddings during training.

Bibliography

[1] Sercan Ö Arik, Mike Chrzanowski, Adam Coates, Gregory Diamos, Andrew Gibiansky, Yongguo Kang, Xian Li, John Miller, Andrew Ng, Jonathan Raiman, et al. "Deep voice: Real-time neural text-to-speech". In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org. 2017, pp. 195–204.

[2] Yoshua Bengio, Paolo Frasconi, and Patrice Simard. "The problem of learning long-term dependencies in recurrent networks". In: *IEEE international conference on neural networks*. IEEE. 1993, pp. 1183–1188.

[3] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. "Learning long-term dependencies with gradient descent is difficult". In: *IEEE transactions on neural networks* 5.2 (1994), pp. 157–166.

[4] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. "A neural probabilistic language model". In: *Journal of machine learning research* 3.Feb (2003), pp. 1137–1155.

[5] Kamil Bennani-Smires, Claudiu Musat, Andreea Hossmann, Michael Baeriswyl, and Martin Jaggi. "Simple Unsupervised Keyphrase Extraction using Sentence Embeddings". In: *Proceedings of the 22nd Conference on Computational Natural Language Learning*. 2018, pp. 221–229.

[6] David M Blei, Andrew Y Ng, and Michael I Jordan. "Latent dirichlet allocation". In: *Journal of machine Learning research* 3.Jan (2003), pp. 993–1022.

[7] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. "Enriching Word Vectors with Subword Information". In: *CoRR* abs/1607.04606 (2016). arXiv: 1607.04606. URL: http://arxiv.org/abs/1607.04606.

[8] Antal Van den Bosch, Toine Bogers, and Maurice De Kunder. "Estimating search engine index size variability: a 9-year longitudinal study". In: *Scientometrics* 107.2 (2016), pp. 839–856.

[9] Florian Boudin. "PKE: an open source python-based keyphrase extraction toolkit". In: *Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: System Demonstrations*. Osaka, Japan, Dec. 2016, pp. 69–73. URL: http://aclweb.org/anthology/C16-2015.

[10] Florian Boudin. "Unsupervised Keyphrase Extraction with Multipartite Graphs". In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*. New Orleans, Louisiana: Association for Computational Linguistics, June 2018, pp. 667–672. DOI: 10.18653/v1/N18-2105. URL: https://www.aclweb.org/anthology/N18-2105.

[11] Adrien Bougouin, Florian Boudin, and Béatrice Daille. "Topicrank: Graph-based topic ranking for keyphrase extraction". In: *International Joint Conference on Natural Language Processing (IJCNLP)*. 2013, pp. 543–551.

[12] Leo Breiman. "Bagging predictors". In: *Machine Learning* 24.2 (Aug. 1996), pp. 123–140. ISSN: 1573-0565. DOI: 10.1007/BF00058655. URL: https://doi.org/10.1007/BF00058655.

[13] John S Bridle. "Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition". In: *Neurocomputing*. Springer, 1990, pp. 227–236.

[14] Denny Britz, Anna Goldie, Minh-Thang Luong, and Quoc V. Le. "Massive Exploration of Neural Machine Translation Architectures". In: *CoRR* abs/1703.03906 (2017). arXiv: 1703.03906. URL: http://arxiv.org/abs/1703.03906.

[15] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. "SMOTE: synthetic minority over-sampling technique". In: *Journal of artificial intelligence research* 16 (2002), pp. 321–357.

[16] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation". In: *CoRR* abs/1406.1078 (2014). arXiv: 1406.1078. URL: http://arxiv.org/abs/1406.1078.

[17] Ben Choi and Zhongmei Yao. "Web page classification". In: *Foundations and Advances in Data Mining*. Springer, 2005, pp. 221–274.

[18] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. "Empirical evaluation of gated recurrent neural networks on sequence modeling". In: *arXiv preprint arXiv:1412.3555* (2014).

[19] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. "Natural language processing (almost) from scratch". In: *Journal of Machine Learning Research* 12.Aug (2011), pp. 2493–2537.

[20] Corinna Cortes and Vladimir Vapnik. "Support-vector networks". In: *Machine learning* 20.3 (1995), pp. 273–297.

[21] Timothy C Craven. "HTML tags as extraction cues for web page description construction". In: *Informing Science* 6 (2003), pp. 1–12.

[22] Yuchen Fan, Yao Qian, Feng-Long Xie, and Frank K Soong. "TTS synthesis with bidirectional LSTM based recurrent neural networks". In: *Fifteenth Annual Conference of the International Speech Communication Association*. 2014.

[23] John R Firth. "A synopsis of linguistic theory, 1930-1955". In: *Studies in linguistic analysis* (1957).

[24] Corina Florescu and Cornelia Caragea. "Positionrank: An unsupervised approach to keyphrase extraction from scholarly documents". In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Vol. 1. 2017, pp. 1105–1115.

[25] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. "Deep sparse rectifier neural networks". In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. 2011, pp. 315–323.

[26] Yoav Goldberg. "Neural network methods for natural language processing". In: *Synthesis Lectures on Human Language Technologies* 10.1 (2017), p. 60.

[27] Edouard Grave, Piotr Bojanowski, Prakhar Gupta, Armand Joulin, and Tomas Mikolov. "Learning Word Vectors for 157 Languages". In: *Proceedings of the International Conference on Language Resources and Evaluation (LREC 2018)*. 2018.

[28] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. "Speech recognition with deep recurrent neural networks". In: *2013 IEEE international conference on acoustics, speech and signal processing*. IEEE. 2013, pp. 6645–6649.

[29] Alex Graves and Jürgen Schmidhuber. "Framewise phoneme classification with bidirectional LSTM and other neural network architectures". In: *Neural Networks* 18.5-6 (2005), pp. 602–610.

[30] Alex Graves, Marcus Liwicki, Santiago Fernández, Roman Bertolami, Horst Bunke, and Jürgen Schmidhuber. "A novel connectionist system for unconstrained handwriting recognition". In: *IEEE transactions on pattern analysis and machine intelligence* 31.5 (2009), pp. 855–868.

[31] Khaled M Hammouda and Mohamed S Kamel. "Efficient phrase-based document indexing for web document clustering". In: *IEEE Transactions on Knowledge & Data Engineering* 10 (2004), pp. 1279–1296.

[32] Kazi Saidul Hasan and Vincent Ng. "Automatic keyphrase extraction: A survey of the state of the art". In: *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Vol. 1. 2014, pp. 1262–1273.

[33] Trevor Hastie, Robert Tibshirani, and JH Friedman. *The elements of statistical learning: data mining, inference, and prediction*. 2009.

[34] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep Residual Learning for Image Recognition". In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016.

[35] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-term Memory". In: *Neural computation* 9 (Dec. 1997), pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735.

[36] Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, and Jürgen Schmidhuber. *Gradient flow in recurrent nets: the difficulty of learning long-term dependencies*. In: *A field guide to dynamical recurrent neural networks. IEEE Press*, 2001.

[37] David H Hubel and Torsten N Wiesel. "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex". In: *The Journal of physiology* 160.1 (1962), pp. 106–154.

[38] Anette Hulth. "Improved automatic keyword extraction given more linguistic knowledge". In: *Proceedings of the 2003 conference on Empirical methods in natural language processing*. Association for Computational Linguistics. 2003, pp. 216–223.

[39] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *International Conference on Machine Learning*. 2015, pp. 448–456.

[40] Kazuki Irie, Zoltán Tüske, Tamer Alkhouli, Ralf Schlüter, and Hermann Ney. "LSTM, GRU, Highway and a Bit of Attention: An Empirical Overview for Language Modeling in Speech Recognition." In: *Interspeech*. 2016, pp. 3519–3523.

[41] Thorsten Joachims. "Text categorization with support vector machines: Learning with many relevant features". In: *European conference on machine learning*. Springer. 1998, pp. 137–142.

[42] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. "Bag of tricks for efficient text classification". In: *arXiv preprint arXiv:1607.01759* (2016).

[43] Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. "An empirical exploration of recurrent network architectures". In: *International Conference on Machine Learning*. 2015, pp. 2342–2350.

[44] Min-Yen Kan. "Web page classification without the web page". In: *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*. 2004, pp. 262–263.

[45] Min-Yen Kan and Hoang Oanh Nguyen Thi. “Fast webpage classification using URL features”. In: *Proceedings of the 14th ACM international conference on Information and knowledge management*. ACM. 2005, pp. 325–326.

[46] Su Nam Kim, Olena Medelyan, Min-Yen Kan, and Timothy Baldwin. “Semeval-2010 task 5: Automatic keyphrase extraction from scientific articles”. In: *Proceedings of the 5th International Workshop on Semantic Evaluation*. 2010, pp. 21–26.

[47] Yoon Kim. “Convolutional Neural Networks for Sentence Classification”. In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, 2014, pp. 1746–1751. DOI: `10.3115/v1/D14-1181`.

[48] Diederik P Kingma and Jimmy Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).

[49] Oh-Woog Kwon and Jong-Hyeok Lee. “Text categorization based on k-nearest neighbor approach for web site classification”. In: *Information Processing & Management* 39.1 (2003), pp. 25–44.

[50] Quoc Le and Tomas Mikolov. “Distributed representations of sentences and documents”. In: *International conference on machine learning*. 2014, pp. 1188–1196.

[51] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. “Backpropagation applied to handwritten zip code recognition”. In: *Neural computation* 1.4 (1989), pp. 541–551.

[52] Yang Li and Tao Yang. “Word embedding for understanding natural language: A survey”. In: *Guide to Big Data Applications*. Springer, 2018, pp. 83–104.

[53] Marina Litvak and Mark Last. “Graph-based keyword extraction for single-document summarization”. In: *Proceedings of the workshop on Multi-source Multilingual Information Extraction and Summarization*. Association for Computational Linguistics. 2008, pp. 17–24.

[54] Marina Litvak, Mark Last, Hen Aizenman, Inbal Gobits, and Abraham Kandel. “DegExt - A language-independent graph-based keyphrase extractor”. In: *Advances in Intelligent Web Mastering–3*. Springer, 2011, pp. 121–130.

[55] Christopher Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven Bethard, and David McClosky. “The Stanford CoreNLP natural language processing toolkit”. In: *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*. 2014, pp. 55–60.

[56] Luís Marujo, Anatole Gershman, Jaime Carbonell, Robert Frederking, and João P. Neto. “Supervised topical key phrase extraction of news stories using crowdsourcing, light filtering and co-reference normalization”. In: *Proceedings of the Eighth International Language Resources and Evaluation (LREC*. 2012.

[57] Masakazu Matsugu, Katsuhiko Mori, Yusuke Mitari, and Yuji Kaneda. “Subject independent facial expression recognition with robust face detection using a convolutional neural network”. In: *Neural Networks* 16.5-6 (2003), pp. 555–559.

[58] Bryan McCann, James Bradbury, Caiming Xiong, and Richard Socher. “Learned in translation: Contextualized word vectors”. In: *Advances in Neural Information Processing Systems*. 2017, pp. 6294–6305.

[59] Rada Mihalcea and Paul Tarau. “Textrank: Bringing order into text”. In: *Proceedings of the 2004 conference on empirical methods in natural language processing*. 2004.

[60] Tomas Mikolov, Edouard Grave, Piotr Bojanowski, Christian Puhrsch, and Armand Joulin. "Advances in Pre-Training Distributed Word Representations". In: *Proceedings of the International Conference on Language Resources and Evaluation (LREC 2018)*. 2018.

[61] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. "Distributed representations of words and phrases and their compositionality". In: *Advances in neural information processing systems*. 2013, pp. 3111–3119.

[62] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. "Efficient estimation of word representations in vector space". In: *arXiv preprint arXiv:1301.3781* (2013).

[63] Frederic Morin and Yoshua Bengio. "Hierarchical probabilistic neural network language model." In: *Aistats*. Vol. 5. Citeseer. 2005, pp. 246–252.

[64] Michael C Mozer. "A Focused Backpropagation Algorithm for Temporal Pattern Recognition". In: *Complex Systems* 3 (1989), pp. 349–381.

[65] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. *The PageRank citation ranking: Bringing order to the web.* Tech. rep. Stanford InfoLab, 1999.

[66] Matteo Pagliardini, Prakhar Gupta, and Martin Jaggi. "Unsupervised Learning of Sentence Embeddings using Compositional n-Gram Features". In: *NAACL 2018 - Conference of the North American Chapter of the Association for Computational Linguistics*. 2018.

[67] Jeffrey Pennington, Richard Socher, and Christopher 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.

[68] Matthew Peters, Waleed Ammar, Chandra Bhagavatula, and Russell Power. "Semi-supervised sequence tagging with bidirectional language models". In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Vancouver, Canada: Association for Computational Linguistics, July 2017, pp. 1756–1765. DOI: `10.18653/v1/P17-1161`. URL: `https://www.aclweb.org/anthology/P17-1161`.

[69] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. "Deep contextualized word representations". In: *Proc. of NAACL*. 2018.

[70] Vijay V Raghavan and SK Michael Wong. "A critical analysis of vector space model for information retrieval". In: *Journal of the American Society for information Science* 37.5 (1986), pp. 279–287.

[71] Prajit Ramachandran, Barret Zoph, and Quoc V Le. "Searching for activation functions". In: *arXiv preprint arXiv:1710.05941* (2017).

[72] Daniele Riboni. "Feature selection for web page classification". In: *Proceedings of the Workshop on Web Content Mapping: A Challenge to ICT*. 2002.

[73] Frank Rosenblatt. *Principles of neurodynamics. perceptrons and the theory of brain mechanisms*. Tech. rep. Cornell Aeronautical Lab Inc. Buffalo NY, 1961.

[74] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning representations by back-propagating errors". In: *Cognitive modeling* 5.3 (1988), p. 1.

[75] Gerard Salton and Christopher Buckley. "Term-weighting approaches in automatic text retrieval". In: *Information processing & management* 24.5 (1988), pp. 513–523.

[76] Mike Schuster and Kuldip K Paliwal. "Bidirectional recurrent neural networks". In: *IEEE Transactions on Signal Processing* 45.11 (1997), pp. 2673–2681.

[77] Karen Sparck Jones. "A statistical interpretation of term specificity and its application in retrieval". In: *Journal of documentation* 28.1 (1972), pp. 11–21.

[78] Carolin Strobl, James Malley, and Gerhard Tutz. "An introduction to recursive partitioning: rationale, application, and characteristics of classification and regression trees, bagging, and random forests." In: *Psychological methods* 14.4 (2009), p. 323.

[79] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. "Sequence to Sequence Learning with Neural Networks". In: *CoRR* abs/1409.3215 (2014). arXiv: `1409.3215`. URL: `http://arxiv.org/abs/1409.3215`.

[80] Stamatina Thomaidou and Michalis Vazirgiannis. "Multiword keyword recommendation system for online advertising". In: *2011 International Conference on Advances in Social Networks Analysis and Mining*. IEEE. 2011, pp. 423–427.

[81] Tijmen Tieleman and Geoffrey Hinton. "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude". In: *Coursera: Neural networks for machine learning* 4.2 (2012), pp. 26–31.

[82] Elena Tutubalina and Sergey Nikolenko. "Demographic Prediction Based on User Reviews about Medications". In: *Computacion y Sistemas* 21 (June 2017), pp. 227–241. DOI: `10.13053/CyS-21-2-2736`.

[83] Xiaojun Wan and Jianguo Xiao. "Single Document Keyphrase Extraction Using Neighborhood Knowledge." In: *AAAI*. Vol. 8. 2008, pp. 855–860.

[84] Paul J Werbos. "Generalization of backpropagation with application to a recurrent gas market model". In: *Neural networks* 1.4 (1988), pp. 339–356.

[85] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alex Smola, and Eduard Hovy. "Hierarchical attention networks for document classification". In: *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 2016, pp. 1480–1489.

[86] Zi Yin and Yuanyuan Shen. "On the dimensionality of word embedding". In: *Advances in Neural Information Processing Systems*. 2018, pp. 887–898.

[87] Shu Zhang, Dequan Zheng, Xinchen Hu, and Ming Yang. "Bidirectional long short-term memory networks for relation classification". In: *Proceedings of the 29th Pacific Asia conference on language, information and computation*. 2015, pp. 73–78.

[88] Xiang Zhang, Junbo Zhao, and Yann LeCun. "Character-level convolutional networks for text classification". In: *Advances in neural information processing systems*. 2015, pp. 649–657.

[89] Yiquing Zhang Zhang, Nur Zincir-Heywood, and Evangelos Milios. "Summarizing web sites automatically". In: *Conference of the Canadian Society for Computational Studies of Intelligence*. Springer. 2003, pp. 283–296.

[90] Peng Zhou, Wei Shi, Jun Tian, Zhenyu Qi, Bingchen Li, Hongwei Hao, and Bo Xu. "Attention-based bidirectional long short-term memory networks for relation classification". In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Vol. 2. 2016, pp. 207–212.

Appendix A

SIC Codes

Table A.1: Full list of all first and second level SIC codes used for classification

Division	SIC	Category name
A. Agriculture, Forestry & Fishing	01	Agricultural Production – Crops
	02	Agricultural Production – Livestock
	07	Agricultural Services
	08	Forestry
	09	Fishing, Hunting & Trapping
B. Mining	10	Metal Mining
	12	Coal Mining
	13	Oil & Gas Extraction
	14	Nonmetallic Minerals, Except Fuels
C. Construction	15	General Building Contractors
	16	Heavy Construction, Except Building
	17	Special Trade Contractors
D. Manufacturing	20	Food & Kindred Products
	21	Tobacco Products
	22	Textile Mill Products
	23	Apparel & Other Textile Products
	24	Lumber & Wood Products
	25	Furniture & Fixtures
	26	Paper & Allied Products
	27	Printing & Publishing
	28	Chemical & Allied Products
	29	Petroleum & Coal Products
	30	Rubber & Miscellaneous Plastics Products
	31	Leather & Leather Products
	32	Stone, Clay & Glass Products
	33	Primary Metal Industries
	34	Fabricated Metal Products
	35	Industrial Machinery & Equipment
	36	Electronic & Other Electric Equipment
	37	Transportation Equipment
	38	Instruments & Related Products
	39	Miscellaneous Manufacturing Industries
E. Transportation & Public Utilities	40	Railroad Transportation
	41	Local & Interurban Passenger Transit
	42	Trucking & Warehousing

	43	U.S. Postal Service
	44	Water Transportation
	45	Transportation by Air
	46	Pipelines, Except Natural Gas
	47	Transportation Services
	48	Communications
	49	Electric, Gas & Sanitary Services
F. Wholesale Trade	50	Wholesale Trade – Durable Goods
	51	Wholesale Trade – Nondurable Goods
G. Retail Trade	52	Building Materials & Gardening Supplies
	53	General Merchandise Stores
	54	Food Stores
	55	Automotive Dealers & Service Stations
	56	Apparel & Accessory Stores
	57	Furniture & Home Furnishings Stores
	58	Eating & Drinking Places
	59	Miscellaneous Retail
H. Finance, Insurance & Real Estate	60	Depository Institutions
	61	Nondepository Institutions
	62	Security & Commodity Brokers
	63	Insurance Carriers
	64	Insurance Agents, Brokers & Service
	65	Real Estate
	67	Holding & Other Investment Offices
I. Services	70	Hotels & Other Lodging Places
	72	Personal Services
	73	Business Services
	75	Auto Repair, Services & Parking
	76	Miscellaneous Repair Services
	78	Motion Pictures
	79	Amusement & Recreation Services
	80	Health Services
	81	Legal Services
	82	Educational Services
	83	Social Services
	84	Museums, Botanical, Zoological Gardens
	86	Membership Organizations
	87	Engineering & Management Services
	88	Private Households
	89	Services, Not Elsewhere Classified
J. Public Administration	91	Executive, Legislative & General
	92	Justice, Public Order & Safety
	93	Finance, Taxation & Monetary Policy
	94	Administration of Human Resources
	95	Environmental Quality & Housing
	96	Administration of Economic Programs
	97	National Security & International Affairs
	98	Zoological Gardens

Appendix B

HTML Preprocessing Tags

Table B.1: Special tags during HTML preprocessing. All style tags are unwrapped, their contents being placed directly in the parent node. Text in the invisible tags is ignored (unless explicitly retrieved, such as the description)

Style tags	Invisible tags
i	style
em	script
b	head
strong	meta
a	
big	
small	
abbr	
acronym	
cite	
strong	
span	
sub	
sup	
var	
tt	