Job Title Classification Strategies for the German Labor Market

Masterthesis

submitted by

Rahkakavee Baskaran

at the

Universität Konstanz

Department of Politics and Public Administration

Center for Data and Methods

1.Gutachter: Prof. Dr. Susumu Shikano

2.Gutachter: JunProf Juhi Kulshresthra

Konstanz, January 6, 2022

Contents

Abbreviations

1 Introduction

2 Related work

2.1 Textclassification

Text classification, a highly researched area, is the process of classifying text documents or text segments into a set of predefined classes. During the last decades, researches developed a various number of classifiers. As **?** summarize in their survey of classifiers, we can group the approaches mainly into three groups. The first group contains traditional methods like **NB!** (**NB!**),**SVM!** (**SVM!**)), K-nearest neighbors, Logistic Regressions or Decision Trees (**????**). Deep learning methods like Convolutional Neural Networks (CNN) or Recurrent Neural Networks (RNN), which are currently the most advanced algorithms for NLP, form the second group. The last group consists of ensemble learning techniques like Boosting and Bagging.

Although learning models often outperforms traditional methods in comparative studies (**?**), not all classifier have constantly good results over all applications. In contrast to traditional methods, deep learning models also usually require millions of data to train an effective model (**?**).

A comparative analysis with **MLR!** (**MLR!**), **RF!** (**RF!**) and K-nearest neighbohrs for news text classification indicates a good performance of **MLR!** and **RF!** compared to K-nearest neighbors, wherby **MLR!** performed better then **RF!**. The authors used **TF-IDF!** (**TF-IDF!**). (**?**).

A study showed from biomedical classfication shows best performance of SVM among Random Forest, Naive Bayes with TFIDF (**?**)

A comparison of BERT with traditional methods and tfidf shows clear outperformance of BERT.

2.2 Challenges of job title classification

Domain-specific and multiclass challenge

Each text classification task presents different challenges. One challenge is that domain-specific problems may arise. There is some work that deals with job classification in the English speaking job market. In terms of classifiers, the corresponding work can be categorised into traditional classifiers or deep learning methods. **?** for example, use a KNN classifier in combination with document embedding as feature selection strategy. ? rely on traditional methods as well, by combining a SVM classifier and a KNN classifier for their job recommendation system. In contrast, the approaches of ?, ? and ? are based on Deep Learning methods. From a higher perspective, there is another dividing line between the approaches. As mentioned earlier, job title normalization can be considered as a typical text classification task (???). ? and ?, however, formulate the task as a string representation approach of similar job titles. A last challenge of text classification tasks comes with the number of classes. As ? show in their classification of tissue, multiclass classification is more difficult than binary classification problems. Partly, because most of classification algorithms were designed for binary problems (?). Approaches for multiclassification can be grouped into two types. Binary algorithms can handle multiclassification naturally. This is, for example, the case for Regression, DT, SVM, KNN and NV. The second type is the decomposition of the problem into binary classification tasks (for the different subtypes see ?). The literature so far does not have a clear answer to solve multiclassification problems. Different approaches, like Boosting (?) or CNN (?) are applied. It is noticeable, however, that many works use variations of SVM (???).

2.3 Short Text classification

Another potential issue is the length of input documents for classification. Job titles are clearly short text with often not more than 50 characters. Short texts suffer from sparseness, few word co-occurrences, missing shared context, noisiness and ambiguity. Traditional methods, however, are based on word frequency, high word co-occurrence and context, which is why they often fail to achieve high accuracy for short texts (???). In their overview, ? present three approaches to solve this. First, since short text data often suffers from unlabeled data in the context of online text data, such as Twitter postings, they suggest using semi-supervised approaches. Second, they recommend to use ensemble learning methods, which focus on the sparseness of the data. Third, ? propose feature dimensionality reduction and extraction of semantic relationship methods. Based on the latter more recent work on short text classification criticizes the use of the “Bag of Word” concept for feature representation as it only reflects the appearance of words in the text. Instead, they represent short texts with semantically similar and conceptual information (**???**). Another question concerning the representation of short texts is whether to represent them as dense or sparse vectors. In their comparison between tf-idf/counter vectorizer and the dense vectorizer word2vec and doc2vec, **?** conclude that among the classifiers Naive Bayes, Logisitic Regression and SVM, the sparse vectorizers achieve the highest accuracy. **?**, conversely, see limitations in sparse representation as it cannot capture the context information. In their work, they integrate sparse and dense representation into a deep neural network with Knowledge powered Attention, which outperform state-of-art deep learning methods, like CNN, for Chinese short texts. Concerning the classifiers, there is no consensus approach for short text classification. For traditional approaches **?**'s results indicate that logistic regression and SVM perform best, while KNN seems to achieve best accuracy in **?**'s work. Similar to job title specific work, more recent work prefers deep learning methods, mostly CNN (**?**).

2.4 Implications

From the above discussed literature three consequences are emerging. First appropriate feature selection or vectorization plays a decisive role in the performance. For that reason, I implement several feature extraction techqniques covering sparse and dense vectors. For sparse vectors I vectorize the data with count vectorizer and **TF-IDF!** vectorizer. Word2vec, Doc2vec and BERT embedding built the group of dense vectorization techniques. The discussion of the different techniques will be the first pillar of the comparison.

Second, relying on one classification does not seem a reasonable option. Instead experimenting and exploring different traditional, deep learning and ensemble classifiers allows for identifying the best classifier based on the task and the data. Therefore I use 4 classifier techniques. I fall back to two traditional methods. The literature shows that **SVM!** and **MLR!** are well compatible with other technqiues, which is why I choose both of them. I also include an ensemble techqniue: **RF!**. As a last method I implement a **BERT!** (**BERT!**) classifier since it is the State-of-Art method currently for text classification. The evaluation of different classifications will built the second pillar of the comparision.

Third as mentioned above the focus of the presented three challenges is on short text classification. The literature on short text classification reveals two points. There are different results on whether sparse or dense techqniques are better suited. Testing different sparse and dense vectorization techniques allows to cover for that point. Second most of the solutions include additional knowledge. Therefore, I use for the training of the dense embedding techqniques additional knowledge from the Taxonomy and compare the results to the models without adding the knowledge, which will build the last pillar of the comparision.

3 Job title data and taxonomy

The training data consists of two data sets. The classes are extracted from the first data set, referred to below as the **KldB!** (**KldB!**) dataset. The **KldB!** dataset contains all information of the **KldB!** taxonomy. The second dataset, called job title dataset in the following contains the necessary data from the job titles. In the first part of this chapter a brief explanation about the Taxonomy structure and both datasets is given. The second part contains of a descriptive analysis of the class distribution of the data.

3.1 KldB 2010 Taxonomy

The "**KldB!**" is structured hierarchically with 5 levels. On each level there is a different number of classes. In the following these classes are also referred to as kldb. On level 1 each class has an id of length one with a number from 0 to 9. Table **??** shows the 10 classes of level 1 with their class names. On level 2, then, each of the 10 classes are divided into one or more subclasses having a class id of length 2 with the first digit indicating the class of level 1 and the second digit the class of level 2. An overview of the all 5 levels with an example of classes is given in table **??**. Note that the example in table **??** does not show on level 2 to level 5 all classes. Thus on level 2 there exists also, e.g. the class id 41 with "Mathematik-, Biologie-Chemie- und Physikberufe", which in turn is divided into other classes on level 3 etc.. With this procedure this ultimatley leads to class ids of length 5 on level 5. An occupation can be classified on every level in the Taxonomy. Considering the classes of the example in table **??**, the job title "Java Developer" could be classified on level 5 to the class 43412. From this id, it is also derivable that the jobtitle belongs, for

IDs KldB 2010	Berufsbereich (Level 1)
1	Land-, Forst- und Tierwirtschaft und Gartenbau
2	Rohstoffgewinnung, Produktion und Fertigung
3	Bau, Architektur, Vermessung und Gebäudetechnik
4	Naturwissenschaft, Geografie und Informatik
5	Verkehr, Logistik, Schutz und Sicherhe
6	Kaufmännische Dienstleistungen, Warenhandel, Vertrieb, Hotel und Tourismus
7	Unternehmensorganisation, Buchhaltung, Recht und Verwaltung
8	Gesundheit, Soziales, Lehre und Erziehung
9	Sprach-, Literatur-, Geistes-, Gesellschafts- und Wirtschaftswissenschaften, Medien, Kunst, Kultur und Gestaltung
0	Militär

Table 1: Overview of classes Level 1 - Berufsbereiche (edited after (**?**))

example, on level 3 to the class “Sofwareentwicklung” (**???**)

The **KldB!** contains of two dimension. The first dimension, the so-called ”Berufsfachlichkeit” structures jobs according to their similarity in knowledge, activties and jobs. This is reflected in the first 4 levels. Considering the again the example from above and the job title ”Fullstack PHP-Entwickler” it is reasonable to classify both on level 1 to ”Naturwissenschaft, Geografie and Information”, because both of them are related to computer science. It also make sense to classifiy them for example to 4341, because both are about sofware development. On level 5, then, a second dimension is introduced. the ”Anforderungsniveau”. This dimension gives information on the level of requirement for a job and 4 possible requirements. In table **??** they are summarized. From the class id of job title “Java Developer”, we can see that the job has been assigned to the second requirement level, since the last digit is a 2 (**???**)

Name	Level	Number of classes	Example
Berufsbereiche	1	10	4: Naturwissenschaft, Geografie und Informatik
Berufshauptgruppen	2	37	43: Informatik-, Informations- und Kommunikationstechnologieberufe
Berufsgruppen	3	144	434: Sofwareentwicklung
Berufsuntergruppen	4	700	4341: Berufe in der Sofwareentwicklung
Berufsgattungen	5	1286	43412: Berufe in der Sofwareenetwicklung - fachlich ausgerichtete Tätigkeiten

Table 2: Overview of **KldB!** (edited after (**?**))

With the **KldB!** 2010, an useful and information-rich occupational classification was created for Germany that reflects the current trends in the labor market (**?**). One strength relies in the construction of the **KldB!**. Instead of just in-

Level of requirement	Class ID	Name long	Name short
1	xxxx1	Helfer- und Anlerntätigkeit	Helfer
2	xxxx2	fachlich ausgerichtete Tätigkeiten	Fachkraft
3	xxxx3	komplexe Spezialstentätigkeiten	Spezialist
4	xxxx4	hoch komplexe Tätigkeiten	Experte

Table 3: Overview of Level of requirements on Level 5
(edited after (**?**))

cluding expert knowledge into the Taxonomy the developement process is based on systematical consideration of information about occupations, as well as statistical procedures for taxonomy developement. Furthermore, the taxonomy was reviewed qualitatively several times in relation to professions and revised. Considering the expressiveness, the **KldB!** has some more benefits. Since the taxonmy is quite recent, it reflects new job classes and market trends very adequately. Further, by including the second dimension, the taxonomy provides a powerful tool to organize job titles into simple requirement classes. In addition, the taxonomy also distinguishes between managerial, supervisory, and professional employees, which is also valuable information. Finally, the taxonomy also convinces with the possibility to switch to "**ISCO!** (**ISCO!**)" through its IDS and thus to normalize jobs to a global standard (**?**).

The **KldB!** dataset contains different information related to the structure descriped above. Besides the class label, the level and the title, on level 5 for each kldb some searchwords are given. There are two types of keywords. One is the job titles that match the respective kldb. On the other hand, real search words, with the help of which the associated kldb can be inferred. Therefore, these searchwords are very helpful knowledge for training classificaiton algorithms, because they contain kldb specific words that are also often present in the job titles. The searchwords will be termed additional knowledge in the rest of the paper.

3.2 Job title data

Job Titles can be scraped from the Federal Employment Agency's job board. Employers must provide additional data for each job posting, including the Job Title, as well as a main internal documentation code that indicates a class in the KLDB taxonomy. There is an option to provide alternative documentation codes if more than one KLDB class is being considered. An example snippet of the scraped data is provided in the Appendix. The documentation code is an internal number of the Federal Employment Agency, which can be uniquely assigned to a KLDB, which, as already mentioned, is specified in the taxonomy data for each kldb. A snippet of the matched training data with the KLDB labels is given in

4 Method

4.1 Conceptual overview

4.2 Preprocessing

4.3 Vectorization Techniques

4.3.1 Count vectorizer

The count vectorizer is one of most simple techniques of converting texts to vectors. It belongs to the family of **BOW!** (**BOW!**) models. **BOW!** models are based on vectors, where each dimension is a word from a vocabulary or corpus. The corpus is built by all words across all documents. **BOW!** models have two important properties. First, they do not consider the order of the words, sequences or the grammar, which is why they also called **BOW!**. Second, each word in the corpus is represented by it's own dimension. Thus the vectors contains a lot of zeros, especially for short texts, which is why they belong to the family of sparse vectors (**?**). Assuming, for example, a corpus contains of 1000 words, meaning each text is built only with these 1000 words, the vector for each text has a length of 1000, thus, producing sparse, high-dimensional vectors (**??**)

For the count vectorizer the values for the vector are generated by counting for each text the frequency of the words occuring. Considering a corpus including only the three words "java", "developer" and "python", the titles "java developer" and "python developer" would be encoded as follows:

	java	developer	python
java developer	1	1	0
python developer	0	1	1

Table 4: Encoding with count vectorizer

The table **??** results in the vectors $(1, 1, 0)$ and $(0, 1, 1)$. Note that if the title "java developer" contains, for example, two times the word "java" then the vector would change to (2,1,0). But since this is not a likely case for short text and especially job titles, the count vectorization here is for the most titles similar to one-hot vector encoding, which only considers the occurence of the words, but not the frequency (**??**)

While being one of the most simple techniques, count vectorizer has several limitations. The main downside is that it does not consider information like the semantic meaning of a text, the order, sequences or the context. In other words a lot of information of the text is losen (?). In addition, count vectorizer does not take into account the importance of words in terms of a higher weighting of rare words and a lower weighting of frequent words across all documents. (?)

4.3.2 TFIDF vectorizer

TF-IDF! belongs like the count vectorizer, to the family of **BOW!** and is as well a sparse vector. In contrast to count vectorizer, it considers the importance of the words by using the **IDF!** (**IDF!**). The main idea of **TF-IDF!** is to produce high values for words which occur often in a documents, but are rare over all documents. The **TF!** (**TF!**) represents the frequency of a word t in a document d and is denoted by $tf(t, d)$. The **DF!** (**DF!**), denoted by df, quantifies the occurence of a word over all documents. By taking the inverse of **DF!** we get the **IDF!**. Intuitvely the **IDF!** should quantify how distinguishable a term is. If a term is frequent over all documents it is not usful for the distinction between documents. Thus the **IDF!** produces low values for common words and high values for rare terms and is calculated as follows (??):

$$idf(t) = log\frac{N}{df}$$

where N is the set of documents. The log is used to attenuate high frequencies. If a term occurs in every document, so $df = N$ the **IDF!** takes a value of 0 ($log(\frac{N}{N})$) and if a term is occuring only one time over all documents, thus distinguish perfectly the document from other documents, $idf(t) = log(\frac{N}{1}) = 1$. (?) Note that there are slight adjustments to calculate the **IDF!**. (?) The implementation of sklearn package, which is used in this work uses an adapted calculation (?):

$$idf(t) = log\frac{1+n}{1+df(t)} + 1$$

Given the $idf(t)$ and tf the **TF-IDF!** can be obtained by multiplying both metrics. The implementation of sklearn normalize in addition the resulting **TF-IDF!** vectors v by the Euclidean norm (?):

$$v_{norm} = \frac{v}{||v||_2}$$

Although **TF-IDF!** considers the importance of words compared to count vectorizer, as a **BOW!** model it suffers from the same limitation as count vectorizer of not taking semantic, grammatic, sequences and context into account (?).

4.3.3 Word2Vec

In contrast to the sparse techniques method mentioned above, word embedding techniques are another popular approach for vectorization. Word embedding vectors are characterised by low dimensions, dense representation and a continous space. They are usually trained with neural networks (??).

Word2Vec, introduced by ?, is one computionally efficient word embedding implementation. The main idea of Word2Vec is based on the distributional hypothesis, which states that similar words appear often in similar context (?). Thus, Word2Vec learns with the help of the context representations of words, which includes the semantic meaning and the context. In such a way it words which are similar are encoded similarily (?).

There exists two variants of Word2Vec. The first variant is based on **BOW!**, the so-called **CBOW!** (**CBOW!**), which predicts a word based on surrounding words. In contrast, the second variant, Skip-Gram, predicts the context words from a word (??). Since in this work a pretrained model from Google is used, which is trained with CBOW, in the following the focus is on **CBOW!**.

CBOW! Word2vec is a 2-layer neural network with a hidden layer and a output softmax layer, which is visualized in figure ??. The goal is to predict a word, the so-called target word, by using the target word's context words. The number of context words is defined by a certain window size. If the window size is 2, the 2 words before and the 2 two words after the target word are considered. Given a vocabulary V, which is a the unqiue set of the words from the corpus, each context word c is fed into the neural network, encoded with one-hot encoding of the length of V, building vector x_c. Thus in figure ?? x_{1k}, for example, could be a one-hot encoded vector of the word before the target word.

The weights between the input layer and the hidden layer are shown in figure ??. Taking the dimension of V and N results in the $V \times N$ matrix W. Given that

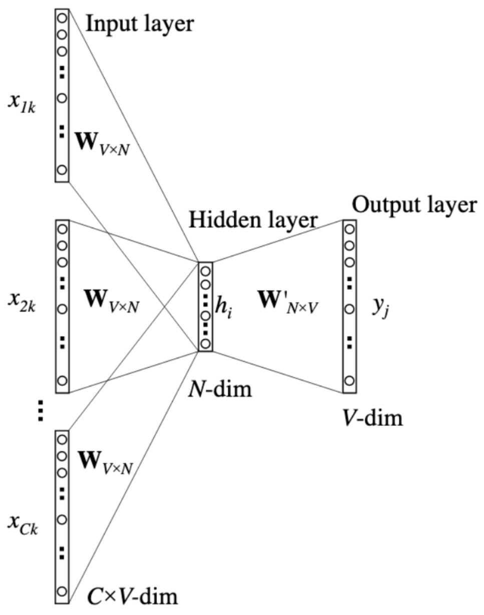

Figure 1: **CBOW!** (**?**, 6)

each row v_w in W represents the weights of the associated word from the input and C equals to the number of context words, the hidden layer h is calculated as follows (**?**):

$$h = \frac{1}{C}W^T(x_1 + x_2 + ... + x_c) = \frac{1}{C}(v_{w_1} + v_{w_2} + ... + v_{w_C})$$

Since the context words are encoded with one hot vector encoding, all except of the respective word value in the vector, which is 1, will be zero. Thus calculating h is just copying the k-th row of matrix W, thus a n-dimensional vector, which explains the second part of the equation. The hidden-layer matrix builds later the word embedding, which is why the decision of the size of the hidden layer defines the dimensions later for the word embedding vector (**?**)

From the hidden to the output layer a $N \times V$ weight matrix is used to calculate scores for each word in V. A softmax function is used to get the word representation. Caluculating the error and using backpropagation the weights are updated respectivley resulting then in a trained neural network. In practice, due to computional efficiency, instead of the softmax function Word2Vec is trained with hierarchical softmax or negative sampling. Both methods are efficient in the sense that they reduce the amount of weight matrix updates. [1]. (**??**).

[1]Since the focus is relying here on the word embeddings from the hidden layer and not the trained neural network itself, no further mathematical details will be given concerning the updating. For

Based on the given theoretical insights, I train two Word2vec models. Both models use a pretrained model from Google and are fine tuned with different data. The first model is fine tuned with the complete dataset. The second model includes the additional knowledge. The model setting is as follows: I use **CBOW!** Word2Vec models with negative sampling technique. The hidden layer size and thus the word embedding vectors is 300, since the vectors have to have the same size as the pretrained Google vectors, which have a size of 300. The minimal count of words is set to 1. The number of times the training data set is iterated, the epoch number, is set to 10. Lastly, the window size for the context is set to 5.

As last step the resulting word embeddings need to be processd in some way to get sentence embeddings for each job title. As the name already indicates Word2Vec cannot output sentence embeddings directly. In order to get the sentence embeddings the word vector embeddings of each job title are averaged.

4.3.4 Doc2vec

Doc2vec, also known as paragraph vectors or Distributed Memory Model of Paragraph Vectors is an extension method of Word2Vec, which outputs directly embeddings for each document (**?**) It was proposed by **?**. Doc2Vec can be used for a variable length of paragraphs, thus it is applicable for bigger documents, but also for short sentences like job titles (**?**).

The main idea is, like for Word2Vec, to predict words in a paragraph. To do so, a paragraph vector and word vectors, like in Word2Vec, for that paragraph are concatenated. The paragraph vector '"acts as memory that remembers what is missing from the current contest - or the topic of the paragraph" (**?**, 3). Therby the paragraph vectors are trained as well with stochastic gradient descent and backpropagation. Similar to Word2Vec practical implementation use hierarchical softmax or negative sampling to fast up training time (**?**).

In figure **??** the algorithm is visualized. As an input the neural networks takes word vectors and a paragraph vector. While the word vectors are shared over all paragraphs, the paragprah vectors are unique for each paragraph. Those paragraphs are represented as unique vectors in a column of a matrix D. The word vectors are respresented in the matrix W as before. In order to predict the word both vectors are combined for example by averaging or concatenating. The Doc2vec implementation

a detailed derivation see (**?**)

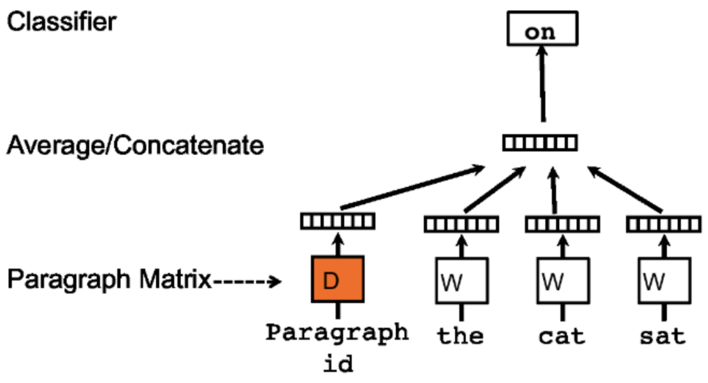

Figure 2: Doc2vec - Distributed Memory Model of Paragraph Vectors (?, 3)

described by ? and also used in this work here concatenate the vectors. Formally this only changes the calculation of h. (?)

Since Doc2vec is a word embedding method it has the same advantages mentioned above in the Word2Vec part. In addition Doc2Vec takes the word order into account. At least in the same way of a large n-gram (?).

Besides the model explained above, Doc2Vec comes also in a second variant, the so-called Distributed Bag of Words of Paragraph model, which ignores the word order. It is not clear which model performs better, although the inventor of Doc2vec propose the first version (?)

Based on this disussion, I created two Distributed Memory Models of Paragraph Vectors. Instead of fine tuning a pretrained model, I trained two custom models, one with the training data and one including the additional knowledge. I set the vector size to 300 with a window size of 5, a minimal count of 1 and trained 10 epochs. Like Word2vec, the models are trained with negative sampling.

4.3.5 BERT

The last vectorization technique, **BERT!**, is the state-of-art language modeling technique developed by (?) at Google. **BERT!** stands out from other langugage models in several ways and outperforms other models for many **NLP!** (**NLP!**) tasks. First, **BERT!** uses bidirectional pretraining. Thus is does not only process from left-to-right or right-to-left, but it merge both of them. Second, it is possible to fine tune the model for specific task without heavily-engineered and computionally costly architectures. Third compared to word2vec it is a context-dependent model. Thus, while word2vec would produce only one word embedding for "'Python" **BERT!** can give based on the context different embeddings. Here ”’Python” as a snake or as a programming language.

Architecture

BERT! uses a multi-layer bidirectional Transformer encoder as the architecture. This transformer architecture was introduced by **?**. It consists of encoder and a decoder and make use of self-attention. Both, encoder and decoder stack include a number of identical layer, each of them including two sub-layers: a Multi-head attention and a feedforward network layer [2] It is out of the scope to elaborate the technical details and implementation of the attention mechanism, which is why in the following a simplified explanation of the attention mechanism is given.

The self-attention mechanism improves the representation of a word, represented by a matrix X, by relating it to all other words in the sentence. In the sentence ”’A dog ate the food because it was hungry”’ (**?**, 10) the self-attention mechanism, for example, could identify by relating the word to all other words, that ”’it” belongs to ”’dog” and not to ”’food”. In order to compute the self attention of a word three additional matrices, the query matrix Q, the key matrix K and a value matrix V are introduced. Those matrices are created by introducing weights for each of them and multiplying those weights with X. Based on those matrices, the dot product between the Q and the K matrix, a normalization and a softmax function are applied in order to calculate an attention matrix Z [3]. **BERT!** uses a multi-head-attention, which simply means that multiple Z attention matrices instead of a single one are used.

BERT! takes one sentence or a pair of sentences as input. To do so it uses WordPiece tokenizer and three special embedding layers, the token, the segement and the position embedding layer for each token, which is visualized in **??**. A WordPiece tokenizes each word which exits in the vocabulary. If a word does not exists words are splitas long a subword matches the vocabulary or a individual character is reached. Subwords are indicated by hashes. For the token embeddings the sentence is tokenized first, then a [CLS] is added at the beginning of the sentence and [SEP] token at the end. For example the job title ‘”java developer” and ‘”

[2]A simplified visualization with a two layer encoder, as well the architecture of a can be found in the Appendix.

[3]For a step-by-step calculating see (**?**)

python developer"' becomes to tokens as shown in the second row of **??**. In order to distinguish between the two titles a segment embedding is added, indicating the sentences. Last the **BERT!** model takes a position embedding layer, which indicates the order of the words. For each token those layers are summed up to get the representation for each token (**??**).

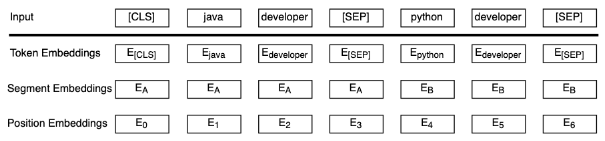

Figure 3: Input **BERT!** (?, 5)

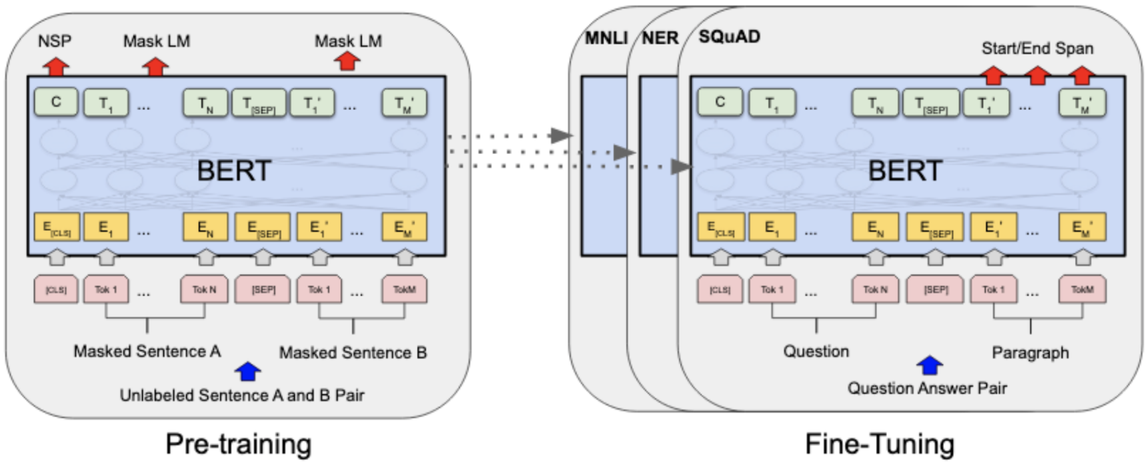

Figure 4: Overview **BERT!** (?, 3)

The **BERT!** algorithm can be described in two steps. First the pretraining phase, which is illustrated on the left-hand side of the figure **??** and the fine-tuning phase, visualized on the right-hand side of figure **??**. The pretraining phase consists of two jobs: Masked language modeling and next sentence prediction.

Pretraining

Masked language modeling means that a percentage of the input tokens are masked at random. For example the job title '"python developer" could be masked as follows: [[CLS] python [MASK] [SEP]]. Since in fine tuning tokens are not masked a mismatch would occur between fine tuning and pretraining, which is why not all of the masked tokens are actually matched with a [mask] token, but also with random token or the real tokens [4]. Instead of predicting the complete sentence, BERT trains to predict the masked tokens. The prediction is performed with a feed forward network and a softmax activation (**??**).

The second task takes again two sentences, but predict whether the second sentence follows the first one. This helps to understand the relationship between the sentences. Each sentence pair is labelled with either isNext or NotNext. By using the [CLS] token, which has the aggregating representation of all tokens, a classification task of whether a sentence pair is isNext or NotNext can be carried out (**??**)

The pretraining of **BERT!** is in contrast to the fine tuning process computionally expensive. Therefore, **?** initially developed different sizes of **BERT!** like **BERT!**-base and **BERT!**-large. Besides those two models, there are plenty of pretrained **BERT!** models for the German case, like **BERT!**-base-german-cased or distilbert-base-german-cased [5]. In an evaluation of German pre-trained language models, (**?**) conclude that the bert-base-german-dbmd-uncased algorithm works quite well. Following their results and own tests on different mode bert-base-german-dbmd-uncased seems to have the best result, which is why I use it for the fine tuning process. The model consists of 12 encoder layers, denoted by L, 12 attention heads, denoted by A and 768 hidden units, which results in total in 110 million parameters. It was trained with 16GB of German texts.

Fine-Tuning

The second phase, the fine-tuning, can be performed in different ways, also depending on the task. For text classifcation there are two main strategies. Either the weights of the pretraind model are updated during the classifcation process. Or the pretrained model is first fine-tuned and then used as a feature extractor. Such it can be then in turn used for, for example, calculating similarities or as an input for classification algorithms.

I train two models with **BERT!**. While the first model includes a classifcation layer, in the following named as **BERT!** classifier, the second model applies **BERT!** as a feature extraction method, in the following named **BERT!** vectorizer.

The **BERT!** classifier is fine tuned with the complete training data set. Prac-

[4]There are specific rules of how to mask. See **?** for detailed implementation

[5]All german **BERT!** are open source and are accessible through the transformers library (**?**)

tically this is done by converting the sentences of the dataset to the appropriate data format as described above and train it with the supervised dataset on some epochs, which then outputs the labels. From a theoretical point of view the last hidden state of the [CLS] token with the aggregation of the whole sentence is used for the classifcation. In order to get the labels **BERT!** uses a softmax function [6] (**?**). In the literature it is not well-understood so far, what exactly happens during the fine-tuning. An analysis of **?** indicates that the fine tuning process is relatively conservative in the sense that they affect only few layers and are specific for examples of a domain. Note that this analysis focussed on other nature language processing task than text classifcation. The set-up of the training is as follows: Testing different epoch numbers indicates that lower epoch size have better results for the model, which is why I fine tune in 6 epochs. For the optimization an adam algorithm, a gradient based optimizer (**?**), with a learning rate of $1e^{-5}$ is used.

In order to get sentence embeddings different strategies, like averaging the output layer of **BERT!** or using the [CLS] token, are applied. Another method, developed by **?** is Sentence-**BERT!**, which is computionally efficient and practicable to implement. Thus, it faciliate to encode sentences directly into embeddings, which is why I use it for the **BERT!** vectorizer. The model is constructed with the bert-base-german-case model and a pooling layer. The pooling layer is added to output the sentence embeddings. The fine tuning process uses a siamese network architecture to update the weights. **??** shows the architecture. The network takes pairs of sentences with a score as an input. Those scores indicate the similarity between the sentences. The network update the weights by passing the sentences through the network, calculating the cosine similarity and comparing to the similarity score (**?**). I create from the training data set and from the kldb taxonomy pairs of similar and dissimilar job titles or searchwords. Similar pairs are pairs from the same kldb class. As a score I choose 0.8. Dissimilar pairs are defined as pairs which are not from the same class. The score is 0.2. Building all combinations of titles and the searchwords for each class would results in a huge dataset. For example class 1 of the training data set has 2755 job titles. Thus we would have already have $\binom{2755}{2} = 3793635$ examples. Since this is computionally to expensive I randomly choose pairs of job titles. For the similar ones I choosed for job titles and for searchwords 3250 pairs. For the dissimilar pairs I choosed respectivley 1750 pairs. Thus, in total, I fine tuned

[6]The explanation of the softmax function follows in the chapter of the classifiers

the model with 10000 examples in 3 epochs.

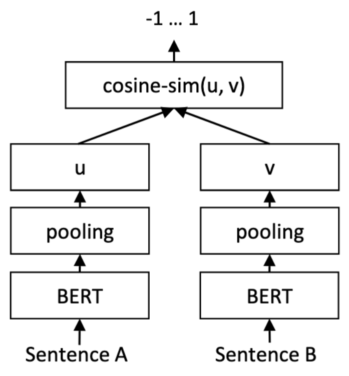

Figure 5: Sentence-**BERT!** siamese architecture (**?**, 3)

4.4 Dimensionality reduction

Dimensionality reduction techniques like **PCA!** (**PCA!**) play an important role in reducing computation time and saving computing resources (**?**). Figure **??** shows the running time of all three classifiers with different data sizes and with and without **PCA!** dimensionality reducation.

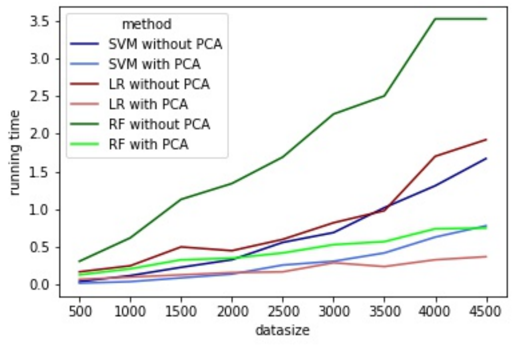

Figure 6: Running time of word2vec with different data sizes

The input of the classifiers are the word2vec word embeddings without the additional information. The bright lines show the running times without dimensionality reduction, while the dark colored lines report the running time with **PCA!** transformation. It becomes clear that the runtime for the transformed embeddings are generally lower. While the magnitude of the differences is almost irrelevant for a data set of 500, the runtime of the non-transformed embeddings increases considerably with the size of the data set for all classifiers. This is most evident with

RF!. Although the runtime of the transformed embeddings also increases for all classifiers, it does so at a much slower pace. Therefore, it can be concluded that the transformation clearly contributes to keeping the runtime lower for large data sets. As already described in chapter x, the training data set is fairly large, which is why it is reasonable to reduce the dimensions.

PCA!, one of the most popular technique for dimensionality reduction, aims to reduce a high-dimensional feature space to a lower subspace while capturing the most important information (**??**). The main idea is to use linear combinations of the original dimensions, so called principal components, to reduce the dimensional space (**??**).

Conceptually in the first step the covariance matrix for the word embeddings, is obtained. The covariance matrix, denoted by **X**, captures the linear relationships between the features of the word embeddings. In a next step the eigenvectors of **X** are calculated. The eigenvector of X defined as (**?**):

$$\mathbf{X}z = \lambda z$$

where z is the eigenvector and λ the eigenvalue. In order decompose **X** to get the eigenvalue Singular Value Decomposition is applied. The eigenvalues are then sorted from highest to lowest and the most significant components n are kept. To transform the data a feature vector in generated. This vector contains of the most n significant eigenvalues. After transposing the mean-adjusted the word embedding and the feature vector, the embeddings can be transformed by multiplying both transposed vectors (**?**).

4.5 Classifier

As pointed out in the literature review **NB!**, **MLR!** and **SVM!** have several advantages for text classifcation tasks. In the following based on a theoretical discussion of each classifier, the exact modeling of the classifiers is justified. The focus and the depth of the explanations of the classifier's characteristics like optimization and decision function, loss or regularization depends on the complexity and the need of explanation in order to understand the basic principle of the classifiers and thus are not all equally structured.

4.5.1 Logistic Regression

MLR!, a generalized linear model, is one of the most used analytical tools in social and natural science for exploring the relationships between features and categorical outcomes. For solving classification problems it learns weights and a bias(intercept) from the input vector. Figure **??** illustrate the idea of the calculation of **MLR!**. To classify examples first the weighted sum of the input vector is calculated. For multiclassfication the weighted sum has to be calculated for each class. Thus given a $f \times 1]$ feature vector x with $[x_1, x_2, ..., x_f]$, a weight vector w_k with k indicating the class k of set of classes K, a bias vector b_k the weighted sum the dot product of w_k and **x** plus the b_k defines the weighted sum. Representing the weight vectors of each class in a $[K \times f]$ matrix **W**, formally the weighted sum is $\mathbf{W}x + b$. In Figure **??** the blue lines for example are a row in **W** and are the weight vectors related to a class labelled with 1 (**?**).

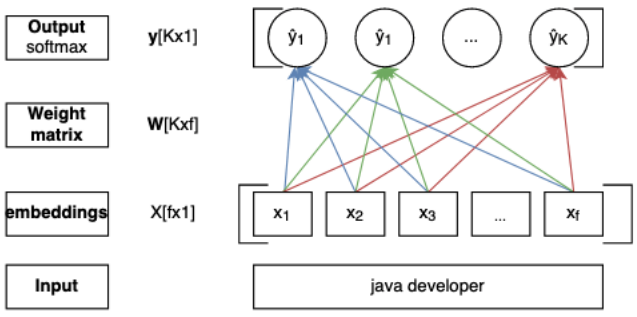

Figure 7: Multinomial Logistic Regression (edit after (**?**, p.))

In a second step the weighted sums are mapped to a value range of $[0, 1]$ in order to actually classify the input. While binary logistic regression uses a sigmoid function to do so, for **MLR!** needs a generalized sigmoid function. This generalization is called the softmax function which outputs probabilities for each of the classes, which is why **MLR!** is also often called softmax regression in the literature. These probabilties models for each class $p(y_k = 1|x)$.

Similar to sigmoid function, but for multiple values the softmax function maps each value of an input vector z with $[z_1, z_2, ..., z_K]$ to a value of the range of $[0, 1]$. Thus outputting a vector of length z. All values together summing up to 1. Formally it is defined as:

$$softmax(z_i) = \frac{exp(z_i)}{\sum_{j=1}^{K} exp(z_j)} \; 1 \leq i \leq K$$

Then the output vector y can be calculated by

$$\hat{y} = softmax(\mathbf{W}x + b)$$

Considering the training of the weights and the bias the goal is to '"maximize the log probability of the true y labels" of the input data. This is commonly done by minimizing a generalized cross-entropy-loss function for **MLR!**. There exists different methods for solving the optimization problem, like stochastic gradient descent or limited-memory Broyden-Fletcher-Goldfarb Shannon solver. The latter converges rapidly and is characterized by a moderate memory, which is why it can converges faster for high-dimensional data (**??**).

For **MLR!** it is common to add regularization parameter. This avoids overfitting and ensures that the model is more generalizable to unseen data. The idea is to penalize weights which have a good classifaction, but use a lot of high weights, more than weights with good classification but smaller weights. There are two popular penality term, the L1 and the L2 penalty. While for L1 the absolute values of the weights are summed and used as the penality term, L2 regualizes with a quadractic function of the weights. With the regularization a parameter C is introduced to control the strength of the regularization, with C being a positive value and smaller C regulizing stronger.

The following setting will be used for the **MLR!**: A **MLR!** with L2 penality with a C value of 1 is used. Since the input vectors, especially for count vectorizer and for **TF-IDF!** are high-dimensional the limited-memory Broyden-Fletcher-Goldfarb-Shannon solver is set for solving. For some trainings converge problems appeared, which is why the maximal iteration of the classifier is set to 10000.

4.5.2 Support Vector Machines

However, **SVM!** also performed well for text classification. Especially for multiclass tasks, as mentioned in the literature review, often different versions of the algorithm are used and showed good performance (**???????**). In general **SVM!** has several advantages for text classifcation. First, text classifcation usually has a high dimensional input space. **SVM!** can handle these large features since they are able to learn independently of the dimensionality of the feature space. In addition **SVM!**s are known to perform well for dense and sparse vectors, which is usually the case for text classification (**?**). Empirical results, for example **?** or **?** confirm the theoretical expectations. It is, therefore, a reasonable option to use a basic version of the **SVM!** algorithm as a baseline.

The general idea of a **SVM!** is to map "the input vectors x into a high-dimensional feature space Z through some nonlinear mapping chosen a priori [...], where an optimal separating hyperplane is constructed" (**?**, 138). In **SVM!** this optimal hyperplane maximizes the margin, which is simply put the distance from the hyperplane to the closest points, so called Support Vectors, across both classes (**?**). Formally, given a training data set with n training vectors $x_i \in R^n, i = 1, ..., n$ and the target classes $y_1, ...y_i$ with $y_i \in \{-1, 1\}$, the following quadratic programming problem (primal) has to be solved in order to find the optimal hyperplane:

$$\min_{w,b} \frac{1}{2} w^T w$$

$$\text{subject to } y_i(w^T \phi(x_i) + b) \geq 1$$

where $\phi(x_i)$ transforms x_i into a higher dimensional space, w corresponds to the weight and b is the bias (**??**) The given optimzation function assumes that the data can be separated without errors. This is not always possible, which is why **?** introduce a soft margin **SVM!**, which allows for missclassfication (**?**). By adding a regularization parameter C with $C > 0$ and the corresponding slack-variable ξ the optimization problem changes to (**??**):

$$\min_{w,b} \frac{1}{2} w^T w + C \sum_{i=1}^{n} \xi_i$$

$$\text{subject to } y_i(w^T \phi(x_i) + b) \geq 1 - \xi_i,$$

$$\xi_i \geq, i = 1, ..., n$$

Introducing Lagrange multipliers α_i and converting the above optimization problem into a dual problem the optimal w meets (**??**):

$$w = \sum_{I=1}^{n} y_i \alpha_i \phi(x_i)$$

with the decision function (**?**):

$$\text{sgn}\ (w^T\phi(x)+b) = sgn(\sum_{i=1}^{n} y_i\alpha K(x_i,x)+b)$$

$K(x_i, x)$ corresponds to a Kernel function, which allows to calculate the dot product in the original input space without knowing the exact mapping into the higher space (**??**).

In order to apply **SVM!** to multiclass problems several approaches have been proposed. One stratetgy is to divide the multi-classifcation problem into several binary problems. A common approach here is the one-against-all method. In this method as many **SVM!** classifiers are constructed as there are classes k. The k-th classifier assumes that the examples with the k label are positive labels, while all the other examples treated as negative. Another popular approach is the one-against-one method. In this approach $k(k-1)/2$ classifiers are constructed allowing to train in each classifier the data of two classes (**?**). Besides dividing the multiclass problem into several binary problems, some researches propose approaches to solve the task in one single optimization problem, like **?**. [7].

In order to find a strong classifier I checked **SVM!**'s with different parameters for the **SVM!**, as well as different multiclass approaches. It appears that a SVM using a soft margin with a $C = 1$ and a one-vs-rest approach has the best results. I also test different kernels, like RBF Kernel or linear kernel. The linear kernel, formally $k(x, x') = x^T x'$, achieved the best results, which is why I choose it for the classifier.

4.5.3 Random Forest Classifier

In contrast to the previous two classifiers **RF!** is a ensemble learning technique. The main idea of ensemble learning techniques is to create a number of learners, classifiers, and combine them. Those learners are for example descision tree or neural networks and are usually homogeneous, which means that each individual learner is based on the same machine learning algorithm. The different ensemble techniques are built on three pillars: the data sampling technique, the strategy of the training and the combination method **??**.

[7]For a detailed overview of all different methods and the method of **?** see **??**

The first pillar, the data sampling is important in sense of that it is not desirable to have same outputs for all classifiers. Thus ensemble techniques need diversity in the output, which means the outputs should be optimally independent and negatively correlated. There are well-established methods for achieving diversity. For example bagging techniques **RF!** falls back to boostrap (**?**). The second pillar rises the question of which techniques should be applied to train the learners of the method. The most popular strategies for the training are bagging and boosting (**?**). The last pillar is about the combining method. Each classifier of the method will output an individual classifcation result and those results have to be combined in some way to achieve an overall result. There are plenty of methods like majority voting or borda count (**?**).

RF! uses as individual classfier decision trees. Before discussing **RF!** in more detail within the three pillars described above, a brief discussion of decision tree is given, in order to understand the mechanism and training procedure of the classifiers.

The main idea of the decision tree algorithm is to '"break up a complex decision into a union of several simple decisions" (**?**, 660) by using trees, with a root node on the top, intermediate nodes and leaf nodes on the bottom. For the root node and each of the intermediate nodes all possible splittings are checked and then are split according to the best feature. Each leaf nodes leads to one of the classification labels. Examples are then classified by traversing the tree from the top to the bottom and choosing at each intermediate node the branch which satisfy the attribute value for the example. The construction of a Decision tree is a recursive procedure (**???**). The algorithm stops for a specific node if all examples of the training set belong to the same class, or if there are no features left for splitting. This might end in tree with a high depth, which is why often pruning is applied to avoid overfitting of the tree **?**.

There are two important points to discuss in constructing. First the types of splitting and second splitting criterion. There are mainly three types of splits: Boolean splits, nominal splits and continous splits. The latter chooses a particular value from the continous feature as the splitting value (**??**). For example considering a word embedding x with 300 dimension and a node t of a decision tree, which is split into a nodes t_{left} and t_{right}. The node t could have the split $x[209] <= 0.336$. Examples with a value smaller than or equal 0.336 at the dimension index 209 of the embedding vector are follow the branch to t_{left}, while all other examples follow the branch to t_{right}.

The splitting criterion is important to identify the best feature for splitting. Intiutively, the criterions should split the data in such a way that leaf nodes are created fast (**?**). There are several measurements, so-called impurity measures to obtain the best split for each node, like gini impurity or information gain. Since **RF!** uses gini impurity, only this criterion will be discussed in detail.

The gini value indicates the purity of a dataset D with n classes. It is defined as follows (**?**, 3156):

$$Gini(D) = 1 - \sum_{i=1}^{n} p_i^2$$

p_i is th probability that a class n occurs in D. The more pure D is the lower the value of the gini value. To determine the best feature k, the dataset is partitioned based on the feature k. For continous features, as in word embeddings, this is done by continous split. Defining V as the total number of subsets and D^v as one of the subsets, the gini impurity for a feature k can be calculated as follows **?**:

$$\text{Gini index}(D, k) = \sum_{v=1}^{V} \frac{|D^v|}{|D|} Gini(D^v)$$

Conceptually the Gini index is the weighted average of the gini value for each subset of D based on a feature k. Thus subsets with more samples are weighted more in the gini index. The optimal feature k^* is then determined by minimizing the Gini impurity over all features K (**?**, 3156):

$$k^* = arg \min_{k \in K} \text{Gini index}(D, k)$$

.

Based on above theoretical explanations of the foundations of decision tree, researchers have developed several algorithms to train decision trees, like Iterative Dichotomiser 3, C4.5. or **CART!** (**CART!**), which is used in **RF!**. **CART!** produces depending on the target variable classification (for categorical variables) or regression trees (for numerical variables). It constructs only binary trees, thus each split is into two nodes. The algorithm uses as impurity measurement gini index and it can handle numerical and categorical input (**?**).

RF! belongs to the family of bagging ensemble techniques. Bagging selects a single algorithm and train a number of independent classifiers. The sampling technique is sampling with replacement (bootstrapping). Bagging combines the individual models by using either majority voting or averaging. **RF!** differentiates from the classic bagging method in the way that it also allows to choose a subset of features for each classifier from which the classifiers can select instead of allowing them to select from the complete range of features (**???**).

Formally **?**, who introduced mainly the **RF!** algorithm defines the classifier as follows:

> ‘”A random forest is a classifier consisting of a collection of tree-structured classifiers $\{h(\mathbf{x}, \Theta_k), k = 1, ...\}$ where the Θ_k are independent identically distributed random vectors and each tree casts a unit vote for the most popular class at the input $\mathbf{x}$.” (**?**, 6).

Θ_k is a random vector which is created for each k-th tree. It is important, that Θ_k is independent of the vectors $\Theta_1 ... \Theta_{k-1}$, thus from all random vectors of the previous classifiers. Although the distribution of the random vectors remain. Combined with the training set, with $\mathbf{x}$ as the input vector a classifier $h(\mathbf{x}, \Theta_k)$ is constructed (**?**). In practical implementation the random component Θ_k is not explicitly used. Instead it is rather used implicitly to generate two random strategies (**?**). The first strategy is the bootstrapping. Thus drawing sample with replacement from the training data set. In order to estimating generalization error, correlation and variable importance **?** applied out-of-bag estimation. Out-of-bag estimation leave out some portion of the training data in each bootstrap. The second strategy is to choose random feature for the splitting. Thus at each node from the set of features only a subset is used to split. While decision trees are often pruned to avoid overfitting, **RF!** does without. The trees grow by applying **CART!**-algorithm. **RF!** uses as combination method for classifcation unweighted voting (**?**).

Based on the above explanations the implemented **RF!** has the following setting: The numbers of learners is 100. Gini is used as the splitting criterion. The maximal number of features is $\sqrt{\text{number of features}}$. Note that sklearn, which is used to implement **RF!** here, uses an optimised algorithm of **CART!**. (**?**).

5 Result

5.1 Evaluation metrics

There exists several metrics for the evaluation of classification approaches in the literature (**?**). The choice of appropriate measurements is a crucial step for obtaining a qualitative comparison in the performance between the baseline algorithms and the new approaches. Often researchers rely on popular metrics like **OA!** (**OA!**). However, especially for multiclass and inbalanced dataset tasks it is difficult to rely only on one measure like **OA!** In order to select appropriate metrics for comparison in the following the most important metrics will be discussed focussing on multiclass classification and imbalanced data sets.

Most metrics rely on a confusion matrix. For the multiclass case this confusion matrix is defined as follows (**?**):

	positive examples			
positive prediction	$c_{1,1}$	$c_{1,2}$	$\dots$	$c_{1,n}$
	$c_{2,1}$	$c_{i,j}$		
	$\vdots$		$\ddots$	$\vdots$
	$c_{n,1}$		$\dots$	$c_{n,n}$

Table 5: Confusion Matrix (edited after (**?**, 113)

From the confusion matrix follows that $c_{i,j}$ defines examples which belong to class j and are predicted as class i. Given that k is the current class, **TP!** (**TP!**) is defined as $tp_k = c_{k,k}$, thus examples which are correctly predicted as the current class k. **FN!** (**FN!**) are defined as those examples which not belonging to the current class k, but are predicted as k. Formally $fn_k = \sum_{i=1,i\neq k}^{n} c_{i,k}$. Next, **TN!** (**TN!**), are examples belonging to the current class m, but are not predicted as m. Formally $tn_k = \sum_{i=1,i\neq k}^{n} \sum_{j=1,j\neq k}^{n} c_{i,j}$. Last, **FP!** (**FP!**) are defined as examples not belonging to class k, but are predicted as such. Formally this can be expressed as: $fp_k = \sum_{i=1,i\neq k}^{n} c_{k,i}$ (**?**)

As mentioned the **OA!** is one of most common metric for performance evaluation. It represents how well the classifier classifies across all classes correctly. Formally, given that N is number of examples and K the number of all classes, this can be expressed as (**?**):

$$OA = \frac{1}{K} \sum_{i=1}^{K} \frac{tp_k + tn_k}{N}$$

Following the formula an accuracy of 1 means that all examples are correctly classified, while a 0 mean that each example is classified with the wrong class. (**?**) Although **OA!** is a widley used metric it is critized for favouring the majority classes, thus not reflecting minority classes appropriatly in unbalanced datasets (**??**)

Two more popular metrics are precision and recall. Precision respresents how well the classifier detects actual positive examples among the positive predicted examples. Recall, also called sensitivity, in contrast, represents how many examples are labelled as positive among the actual positive examples (**?**). For the multiclass scenario, two different calculation approaches for each of the metrics are proposed: micro and macro average (**?**). In the macro approach first the metric is calculated for each class k against all other classes. The average of all of them is built. Formally:

$$precision_{macro} = \frac{1}{K}\sum_{i=1}^{k}\frac{tp_i}{tp_i + fp_i}$$

$$recall_{macro} = \frac{1}{K}\sum_{i=1}^{k}\frac{tp_i}{tp_i + fn_i}$$

In contrast the micro approach aggregates the values, which can be formally expressed as follows:

$$precision_{micro} = \frac{\sum_{i=1}^{K} tp_i}{\sum_{i=1}^{K} tp_i + fp_i}$$

$$recall_{micro} = \frac{\sum_{i=1}^{K} tp_i}{\sum_{i=1}^{K} tp_i + fn_i}$$

There is a trade-off between precision and recall (**?**). The F-measure capture both precision and recall by taking the harmonic mean between both. It is calculated as follows (**??**):

$$F_{micro} = 2 \cdot \frac{precision_{micro} \cdot recall_{micro}}{precision_{micro} + recall_{micro}}$$

$$F_{macro} = 2 \cdot \frac{precision_{macro} \cdot recall_{macro}}{precision_{macro} + recall_{macro}}$$

Apart from the trade-off between recall and precision, there is also a tradeoff between sensitivity and specificty (1- sensitivity). Using a **ROC!** (**ROC!**), which plots the specifity against the sensitivity the trade-off can be visualized for different

	LR	**SVM**	**RF**
CountVectorizer	0.71	0.67	0.63
TFIDF	0.70	0.67	0.62
Word2Vec_I	0.52	0.51	0.60
Word2Vec_II	0.52	0.50	0.60
Doc2Vec_I	0.46	0.43	0.53
Doc2Vec_II	0.44	0.41	0.51
BERT	0.76	0.77	0.76

Table 6: Evaluation of Level 1 classification - Accuracy

	LR	**SVM**	**RF**
CountVectorizer	p: 0.76, r: 0.63, F1: 0.68	p: 0.74, r: 0.58, F1: 0.63	p: 0.68, r: 0.54, F1: 0.57
TFIDF	p: 0.77, r: 0.61, F1: 0.66	p: 0.74, r: 0.57, F1: 0.62	p: 0.68, r: 0.52, F1: 0.55
Word2Vec_I	p: 0.60, r: 0.39, F1: 0.42	p: 0.51, r: 0.40, F1: 0.41	p: 0.62, r: 0.53, F1: 0.56
Word2vec_II	p: 0.61, r: 0.41, F1: 0.44	p: 0.50, r: 0.41, F1: 0.43	p: 0.60, r: 0.51, F1: 0.54
Doc2Vec_I	p: 0.43, r: 0.33, F1: 0.34	p: 0.40, r: 0.31, F1: 0.32	p: 0.69, r: 0.40, F1: 0.41
Doc2Vec_II	p: 0.43, r: 0.30, F1: 0.31	p: 0.37, r: 0.30, F1: 0.30	p: 0.49, r: 0.37, F1: 0.38
BERT	p: 0.75, r: 0.76, F1: 0.75	p: 0.76, r: 0.76, F1: 0.76	p: 0.79, r: 0.73, F1: 0.74

Table 7: Evaluation of Level 1 classification - macro

thresholds. The area under the curve then can be used to obtain the performance of the classifier. A large area indicates a better classifier (**??**).

As shown above, there are several metrics for evaluating the performance of a classifier, with the metrics having different focuses. Since the job title classification involves multiclass classification and the descriptive analysis show that the data is clearly unbalanced, at least for some classes in level 5, it is not reasonable to base the evaluation solely on the **OA!**. Taking precision, recall and the harmonic mean into account would capture the performance of the minority classes as well. The **ROC!** curve does gives, due to its visualization a good impression for the performance, but it is not feasible for high number of classes. Following this argumentation the performance of the classifiers will be evaluated with accuracy, precision, recall, F-measure and Cohen's Kappa.

5.2 Experimental results

6 Conclusion and Limitations

7 sentences

? concludes that for there are different results for different classifiers.

	LR	**SVM**	**RF**
CountVectorizer	p: 0.71, r: 0.71, F1: 0.71	p: 0.67, r: 0.67, F1: 0.67	p: 0.63, r: 0.63, F1: 0.63
TFIDF	p: 0.70, r: 0.70, F1: 0.70	p: 0.67, r: 0.67, F1: 0.67	p: 0.62, r: 0.62, F1: 0.62
Word2Vec_I	p: 0.52, r: 0.52, F1: 0.52	p: 0.51, r: 0.51, F1: 0.51	p: 0.60, r: 0.60, F1: 0.60
Word2vec_II	p: 0.52, r: 0.52, F1: 0.52	p: 0.50, r: 0.50, F1: 0.50	p: 0.60, r: 0.60, F1: 0.60
Doc2Vec_I	p: 0.46, r: 0.46, F1: 0.46	p: 0.43, r: 0.43, F1: 0.43	p: 0.53, r: 0.53, F1: 0.53
Doc2Vec_II	p: 0.44, r: 0.44, F1: 0.44	p: 0.41, r: 0.41, F1: 0.41	p: 0.51, r: 0.51, F1: 0.51
BERT	p: 0.76, r: 0.76, F1: 0.76	p: 0.77, r: 0.77, F1: 0.77	p: 0.76, r: 0.76, F1: 0.76

Table 8: Evaluation of Level 1 classification - micro

? The results of doc2vec and word2vec seem in the first place counterintuive. But short text high dimensional space. Also other papers like **?** show similar results for short texts.

A Naive Bayes Classifier

A.1 Theory

The **NB!**, a family of probabilistic classifiers, uses Bayes' rule in order to determine the most likely class for each document (**?**). All **NB!** classifiers rely on the conditional independence assumptions which means, that "features are independent of each other, given the category variable" (**?**, 48). Depending on whether the features are discrete or continous, different distributions, so-called event models are proposed. While from a theoretical perspective for continous features Gaussian distribution is well-suited, for discrete features usually Bernoulli or multinomial distibutions are applied (**?**). Although, popular practical implementations, like the one from sklearn, allow as well fractional counts for multinomial distributions (**?**). Trying different event models, the multinomial **NB!** shows indeed for both Count Vectorizer and for TFIDF the best results, which is why I choose it as event model for the baseline.

The multinomial **NB!** classifies according to the most likely class. Given that a document d has $t = 1, ..., k$ terms and can be assigned to $c = 1, ..., j$ classes, the probability of a term in a document given a class is calculated as (**?**):

$$P(t_k, c_j) = \frac{occurence(t_k, c_j) + 1}{\sum occurence(t, c_j) + |V|}$$

where $|V|$ is the cardinality of the vocabulary. In the denominator 1 is added, so-called Laplace smoothing, in order to avoid zeros, which is the case if the number of terms in a document for one class is zero. (**?**). Further given that $N_c = count(c_j)$ is the number of documents belonging to class c_j and N is number of documents the probability of c_j is defined as $\frac{N_c}{N}$. The probability of a document d belonging to a class c_j can then formulated as follows (**?**, 258):

$$P(c_j|d) \propto P(c_j) \prod_{i=1}^{k} P(t_i|c_j)$$

Then the most likely classes can be determined by (**?**):

$$\arg\max_{c \in C} P(c_j) \prod_{i=1}^{k} P(t_i|c_j)$$

A.2 Results