UNIKASSEL VERSITÄT
Department of Electrical Engineering and Computer Science
Chair Intelligent Embedded Systems

Identification of key information with topic analysis on large unstructured text data

BACHELORTHESIS

Department of Electrical Engineering and Computer Science
University of Kassel

Author Name:	Klara Maximiliane Gutekunst
Address:	*** REMOVED ***
	34125 Kassel
Matriculation number:	*** REMOVED ***
E-Mail:	klara.gutekunst@student.uni-kassel.de
Department:	Chair Intelligent Embedded Systems
Examining board 1:	Prof. Dr. rer. nat. Bernhard Sick
Examining board 2:	Prof. Dr. Gerd Stumme
Supervisor:	Dr. Christian Gruhl
Date:	October 17, 2023

Abstract

Finding relevant documents and connections between multiple ones becomes significantly more difficult due to the sheer amount of documents available. Institutes, such as German tax offices, have access to leak data, for instance, the Bahama leak, containing huge amounts of documents and valuable information yet to be extracted. However, these institutes, companies and individuals do not have sufficient resources to explore individual documents in order to find a specific one or to identify the key topics of them. Hence, computational means, such as text mining or topic modelling, may facilitate the situation. This thesis proposes an approach to finding relevant documents which share common topics from a large text corpus.

Contents

List of abbreviations

CSS	Cascading Style Sheet
LDA	Latent Dirichlet Allocation
TF-IDF	Term Frequency - Inverse Document Frequency
TF	Term Frequency
IDF	Inverse Document Frequency
BERT	Bidirectional Encoder Representations from Transformers
BERTopic	BERT Topic Model
Doc2Vec	Document to Vector
Word2Vec	Word to Vector
CBOW	Continuous-Bag-of-Words
GloVe	Global Vectors
USE	Universal Sentence Encoder
PCA	Principal Component Analysis
kNN	k-nearest neighbor
API	Application Programming Interface
HTML	Hypertext Markup Language
CSS	Cascading Style Sheet
JSON	JavaScript Object Notation
PKL	Pickle
HNSW	Hiercharical Navigable Small World
OPTICS	Ordering Points To Identify the Clustering Structure
AE	Autoencoder
DBSCAN	Density-Based Spatial Clustering of Applications with Noise
HDBSCAN	Hiercharical Density-Based Spatial Clustering of Applications with Noise
KL	Karhonen-Loéve
SVD	singular value decomposition
HTTP	Hypertext Transfer Protocol
URL	Uniform Resource Locator
SQL	Structured Query Language
NoSQL	Not only SQL
ACID	Atomicity, Consistency, Isolation, Durability
VSM	Vector Space Model
NN	Neural Network
DNN	Deep Neural Network
RNN	Recurrent Neural Network

RMSE	Root Mean Square Error
ML	Machine Learning
NLP	Natural Language Processing
PVDM	Paragraph Vector Distributed Memory
PV-DBOW	Distributed Bag of Words
SNLI	Stanford Natural Language Inference
BiLSTM	bi-directional Long Short-Term Memory
LSTM	Long Short-Term Memory
DAN	Deep Averaging Network
BERT	Bidirectional Encoder Representations from Transformers
SBERT	Sentence-BERT
GloVe	Global Vectors for Word Representation
RSME	Root Mean Square Error
CTM	Correlated Topic Model
LDA	Latent Dirichlet Allocation
LSA	Latent Semantic Analysis
PLSA	Probabilitstic Latent Semantic Analysis
UMAP	Uniform Manifold Approximation and Projection
UI	User Interface
GB	Gigabyte
PDF	Portable Document Format

1 Introduction

The Bahamas leak is a collection of roughly 38 Gigabyte (GB) documents, which were leaked in 2016 [40]. Tax offices examine the data to identify tax evasion. However, it has proven to be challenging to identify relevant documents and connections between documents due to the amount of documents in the leak. Therefore, the goal of this thesis is to support the investigators of the tax offices.

The approach proposed in this thesis includes text mining and exploration methods including topic modelling. Documents are grouped based on their semantic similarity or appearance. In this work, a topic is denoted as a set of words which appears more often than average. Hence, a topic is a statistical phenomenon.

Besides literature research, application and evaluation of the methods identified, certain preprocessing methods have proven to be eminent to successful work with unstructured text data. These methods include tokenization, lemmatization and stop-word-lists.

1.1 Motivation

On a broader scope, this thesis aims to provide computational means to facilitate the work with large unstructured text data for individuals. The goal is to actively use Machine Learning (ML) techniques to analyse large text corpus and thus, reduce the amount of manual human work. This thesis assumes that there are similar documents and that it is valuable to explore semantically or visually similar documents simultaneously. Hence, the analysis of the text corpus includes analysis in terms of textual information, i.e. content, and visual information, i.e. appearance and layout, mimicking a human approach. The information gathered ought to be used to identify similarities between documents and group them. Topics of a document cluster do not have to be labelled specifically but should be interpretable.

1.2 Research Questions

In order to support individuals exploring large unstructured text data, this thesis aims to provide computational means to facilitate the work with large unstructured text data. In this work, different methods to derive semantic and visual information from unstructured text data are applied. The techniques ought to be compared and evaluated.

In order to facilitate the exploration of the data, a User Interface (UI) should be provided. This UI should assist a natural human approach to exploration: A human finds a document of interest, for instance, by random sampling, and thus, wants to find similar documents.

In the following, the goals of this work are defined.

Evaluation of scalability. The techniques applied should be evaluated in terms of their usability for large datasets, such as the Bahamas leak.

Similarity of grouped documents. The document cluster should be derived from either semantically or visually similar documents.

Topic identification. The topics identified should be meaningful to the task at hand and human interpretable.

Little latency. The database should be calculated offline to ensure little latency when executing queries.

Usability. The methods should be bundled in an application, which is easy to use and does not require any programming skills.

1.3 Structure of the Thesis

The rest of this thesis is structured as follows. Chapter 3 provides background information on the topic of this thesis. Chapter 4 describes the implementation of the methods. Chapter 5 evaluates the methods. Chapter 6 discusses the results. Chapter 7 concludes this thesis and Chapter 8 gives an outlook on future work.

2 Related work

3 Fundamentals

The following chapter outlines the fundamentals of the methods used in this work. First, the preprocessing of the data is described in section 3.1. Then, the different similarity measurements are introduced in section 3.2. Afterwards, a variety of ways to generate numerical representations of textual data is outlined in section 3.3. Then, section 3.6 presents multiple clustering methods. The database applied is described in section 3.7. Finally, the libraries used to implement the web application are introduced in section 3.8 and section 3.9.

3.1 Preprocessing

Similar to other ML domains, Natural Language Processing (NLP) requires preprocessing of the data. Usually, textual data contains irrelevant information and noise. Hence, preprocessing improves the performance and the results [46]. The next sections describe a selection of the preprocessing steps applied in this work.

3.1.1 Tokenization

Tokenization is the process of splitting a text into smaller pieces, so-called *tokens*. Tokens can be words and punctuation marks [8]. However, the definition of a token depends on the application. For instance, certain tokenization implementations may identify tokens as subsequent series of non-whitespace characters omitting all numbers and punctuation marks [51].

3.1.2 Stemming

In order to avoid language inflections, i.e. treating words with similar meanings differently, stemming is applied [46]. According to Bird et al., *stemming* is the process of striping off any affixes, i.e. prefixes and suffixes [51], from a word and returning the stem. Different types of stemmers are better suited for certain applications than others. Hence, the choice of the stemmer depends on the application.

3.1.3 Lemmatization

Stemming and lemmatization are used to reduce the vocabulary size [46]. By ensuring the resulting stem is a valid word, the process of stemming is called *lemmatization* [8]. Some implementations of lemmatizers only stem words if the result is in its dictionary. Since lemmatizers validate the result prior to returning it, they are usually slower than stemmers [8].

The *WordNetLemmatizer* from the `nltk` package requires a vocabulary. According to Radu et al., it is frequently used for English texts. It considers not only the meaning of words, but also the order of the words [46].

3.1.4 Stop-Word-Removal

Omitting words that are not relevant to the context of the text is called *stop-word-removal*. Stop words not only depend on the domain but also the language [51].

3.1.5 Lower case

Words with capital letters are converted to lowercase.

3.2 Similarity Measurement

Since embeddings represent texts as numerical vectors, they not only facilitate human interpretability of relationships between texts using the text's respective point in a N-dimensional space, but they also enable the use of metrics, i.e. similarity measures, to quantify the similarity between texts [51, 32].

There are several similarity measures, such as the dot product quantifying the number of shared tokens of two texts, the (soft) cosine similarity, which is the normalized dot product and calculates the angle between two vectors, and many more [51, 32, 48]. The following section outlines a few similarity measures.

3.2.1 Euclidian distance

The *euclidian distance* is a distance measure. In order to measure the distance between two points in a N-dimensional space, the root of the sum of squared distances between the respective values of every dimension is calculated. The distance function Euclidean (L2) norm is given in Equation 3.1 c.f. [32]. The points a, b correspond to objects d_1, d_2.

$$d_E(a,b) = \sqrt{\sum_{i=1}^{N}(a_i - b_i)^2} \tag{3.1}$$

3.2.2 Cosine Similarity

In the traditional bag-of-words approach the texts are represented as vectors of TF-IDF coefficients [10]. Without further processing, the vector is of size N, N being the number of different words of the texts [10]. Hence, a vector represents its corresponding text in a N-dimensional space. This space is called Vector Space Model (VSM) [50].

The similarity between two texts is measured by the cosine of the angle between their respective vectors [50]. The cosine similarity is defined in Equation 3.2 from [50]. The dot-product is defined as $a \cdot b = \sum_{i=1}^{N} a_i b_i$. It is normalized with $\|x\| = \sqrt{x \cdot x}$ to unit Euclidean length [50]. The cosine similarity is a value between 0 and 1 for positive values [50]. According to Sidorov et al., the formula has a time and space complexity of $O(N)$ for a pair of N-dimensional vectors.

$$cosine(a,b) = \frac{a \cdot b}{\|a\| \times \|b\|} = \frac{\sum_{i=1}^{N} a_i b_i}{\sqrt{\sum_{i=1}^{N} a_i^2}\sqrt{\sum_{i=1}^{N} b_i^2}} \tag{3.2}$$

The formula Equation 3.2 assumes that the vectors, which span the VSM are orthogonal and thus, completely independent [50]. However, in practical applications, this often is not the case [50].

3.2.3 Soft Cosine Similarity

This similarity measure not only evaluates whether two texts consist of the same words but also takes into account the semantic (word-level) similarity or lexical relation of different words of the texts [10]. Hence, it improves the shortcomings of the traditional cosine

similarity measure, which assumes the tokens of the vocabulary are completely independent of each other [50].

According to Sidorov et al., in order to model this additional information, more dimensions are added to the VSM. These dimensions can be obtained, for instance, by multiplying the mean of two features of one vector with the similarity between them [50]. The similarity can be calculated by using Levenshtein distance, i.e. the number of operations necessary to convert one string into another, or using a dictionary of synonyms [50].

Since this approach no longer assumes that different words are independent of each other, the basis vectors which span the VSM are no longer orthogonal [50]. The formula for the soft cosine similarity is defined in Equation 3.3 from [50]. The similarity s_{ij} between the i-th and j-th basis vector is obtained using a similarity measure, such as synonymy [50].

$$soft\_cosine(a, b) = \frac{\sum_{i=1}^{N} \sum_{j=1}^{N} s_{ij} a_i b_j}{\sqrt{\sum_{i=1}^{N} \sum_{j=1}^{N} s_{ij} a_i a_j} \sqrt{\sum_{i=1}^{N} \sum_{j=1}^{N} s_{ij} b_i b_j}} \tag{3.3}$$

According to Charlet and Damnati, the similarity between two texts is non-zero as soon as they share related words [10]. If there is no similarity between different features, the soft cosine similarity from Equation 3.3 is equal to the cosine similarity from Equation 3.2. The time and space complexity of the soft cosine similarity is $O(N^2)$ [50].

In order to reduce the complexity, Sidorov et al. propose to use a sparse similarity matrix which only stores $s_{ij} > t$, t being a threshold [50].

3.3 Embeddings

Usually, ML techniques embeddings, such as K-Means, require the text input data to be converted to embeddings [36]. Embeddings are numerical representations of words, sentences or texts. They can be used to present the textual data as real-valued vectors in a VSM. VSMs are commonly used due to their conceptual simplicity and because spatial proximity serves as a metaphor for semantic proximity [62, 9, 48, 4]. Representations in a VSM can improve the performance in NLP tasks [38]. According to Zhang et al., when representing text the first step is indexing, i.e. assigning indexing terms to the document. The second task is to assign weights to the terms which correspond to the importance of the term in the document. The weights assigned depend on the method and the assumptions of the model chosen to carry out the assignments.

The following section outlines a selection of embeddings. Let a corpus of documents be denoted $D = \{d_1, d_2, ..., d_M\}$, the number of documents in the dataset $M = \|D\|$, and a sequence of terms w_{ij} or so-called document $d_i = \{w_{i1}, w_{i2}, ..., d_{iV}\}$, V being the length of the vocabulary, i.e. set of distinct words [46].

3.3.1 TF-IDF

TF-IDF provides a numerical representation of a word in a document [46]. It considers the frequency f_{w_{ij},d_i} of a word w_{ij} in a document d_i and the frequency of a word in the whole corpus.

TF-IDF is calculated as displayed in Equation 3.4 from [46] and exemplatory in Figure 3.1. Term Frequency (TF) is computed using $TF(w_{ij}, d_i) = f_{w_{ij},d_i}$, whereas the Inverse Document Frequency (IDF) is computed using $IDF(w_{ij}, D) = \log_2 \frac{M}{M_{ij}}$, M_{ij} being the number of documents the term w_{ij} appears in. IDF measures the importance of a term w_{ij} in the corpus of documents D. The underlying assumption of IDF is that a term's importance to the data corpus is inversely proportional to its occurrence frequency [62]. In other words: Terms which appear in many documents are not as important and thus, weighted less than document-specific terms.

$$TFIDF(w_{ij}, d_i, D) = TF(w_{ij}, d_i) \cdot IDF(w_{ij}, D) \tag{3.4}$$

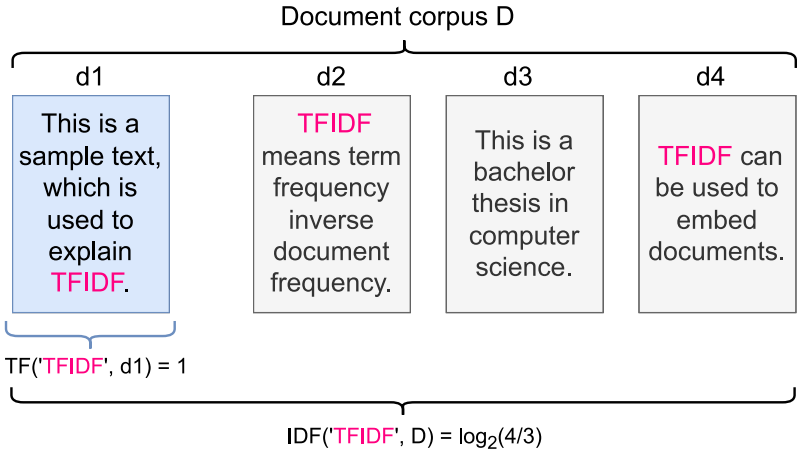


Figure 3.1: Exemplary calculation of TF-IDF for a document corpus D: TF only considers the documents of interest while IDF incorporates the importance of the word with respect to D.

According to Zhang et al., the computation complexity of TF-IDF embeddings is $O(V \cdot M)$ The TF-IDF has several drawbacks [46, 62]:

- TF-IDF does not consider semantic similarities between words.

- TF-IDF does not take into account the order of words in a document.

- TF-IDF often produces high dimensional representations which have to be postprocessed to reduce their dimensionality, e.g., by using Principal Component Analysis (PCA).

- The embeddings are not derived from a mathematical model of term distribution and thus, are occasionally criticised as not well reasoned.

3.3.2 Doc2Vec

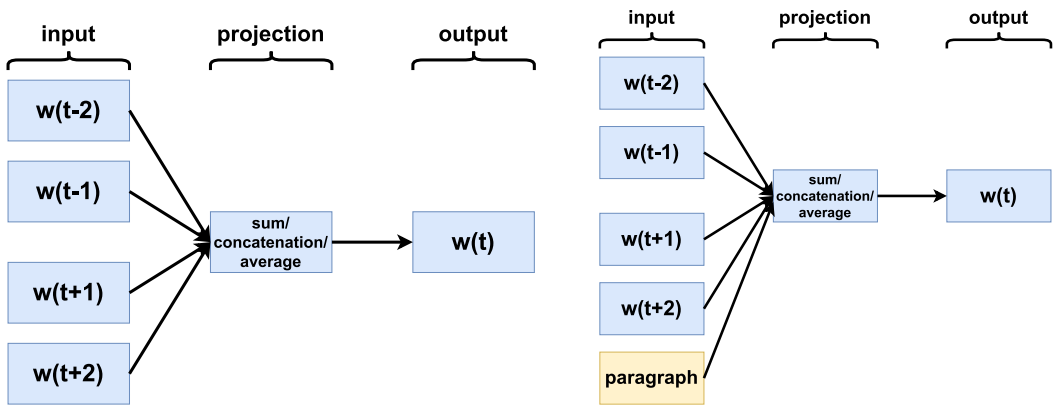
Another term used for Doc2Vec is *Paragraph Vector* [46, 36]. Doc2Vec addresses the problems of TF-IDF by encoding texts as $N-$dimensional vectors learnt using the words' context [46]. Hence, it preserves semantic similarities between words and encodes linguistic regularities and patterns [38]. The model handles inputs of different dimensions and thus, tokens are sentences, paragraphs or documents.

Doc2Vec is an adaption of the Word to Vector (Word2Vec) model, which maps words into a VSM taking into consideration their semantic similarities [46]. Both approaches assume that words appearing in similar contexts are semantically similar. The Word2Vec embedding is obtained using a Neural Network (NN). The NN is shallow, i.e. has only one hidden layer. This hidden layer creates the embedding of input data. There are two approaches to designing the architecture of the NN:

- Paragraph Vector Distributed Memory (PVDM): Predicts a word given a context [36, 37].

- Distributed Bag of Words (PV-DBOW): Predicts the context given a word [33, 38, 36].

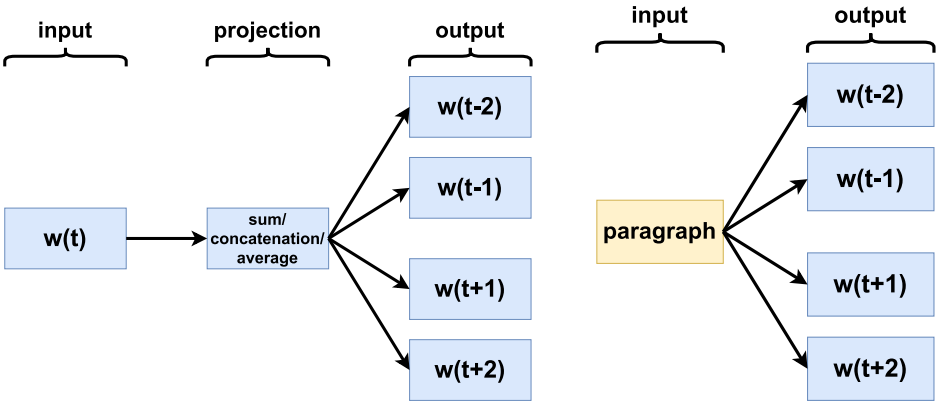
The PVDM extends the CBOW to work on a corpus of documents instead of on a set of words [46]: As usual, vectors representing the words are obtained using the CBOW model. The CBOW has a single-layer architecture [45]. The word vectors can be concatenated, averaged or summed up [36]. Each document is mapped to a vector using an additional document-to-vector matrix. Both document and word vectors are initialized randomly, but trained to convey meaning in terms of semantic differentiation. In order to train the model, centre words are predicted using the context. The context consists of words within a sliding window and their document, respectively represented as vectors [36]. The document vector is added to incorporate the document's topic and thus, acts like a memory [36, 4]. In the work of [36], the document vector is concatenated to the word vectors. The resulting vector is the prediction of the central word. The approaches are displayed in Figure 3.2.



(a) Continuous-Bag-of-Words (CBOW) architecture cf. [37].

(b) PVDM architecture cf. [36].

Figure 3.2: Both approaches predict the centre word using the context. PVDM is an adaption of CBOW to work on a set of documents or paragraphs instead of words.



(a) Skip-gram architecture cf. [37].

(b) PV-DBOW architecture cf. [36].

Figure 3.3: Both approaches predict the context. PV-DBOW is an adaption of Skip-gram to work on a set of documents or paragraphs instead of words.

According to [36], both CBOW and PVDM are trained using stochastic gradient descent and backpropagation. The authors of [36] also state that the document vectors are unique, while the word vectors are shared across the whole corpus.

The PV-DBOW approach is the adaption of the Word2Vec algorithm Skip-Gram and predicts the context given the document [36]. The approaches are displayed in Figure 3.3.

3.3.3 USE

Cer et al. have published their USE model on TF Hub. They propose two architectures, one based on a Transformer and one based on a Deep Averaging Network (DAN). Both models' input is a lowercase tokenized string. Their output is a 512-dimensional vector.

The transformer model is more accurate and more complex than the DAN model [9]. The transformer's (self) attention is used to compute context-aware word embeddings, which consider both the word order and their semantic identity. Since the sequence of word embeddings of a sentence would produce embeddings of different dimensions, the approach postprocesses the word embeddings. A sentence vector is obtained by computing the element-wise sum of the word embeddings and normalizing the result by dividing by the square root of the sentence length.

The DAN model receives real-valued embeddings of words and bi-grams as input. Those input vectors can be obtained from the text strings using models such as the bag of words model [24]. The embeddings are averaged and subsequently passed to a feedforward Deep Neural Network (DNN) [9]. The architecture of the DAN model is depicted in Figure 3.4.

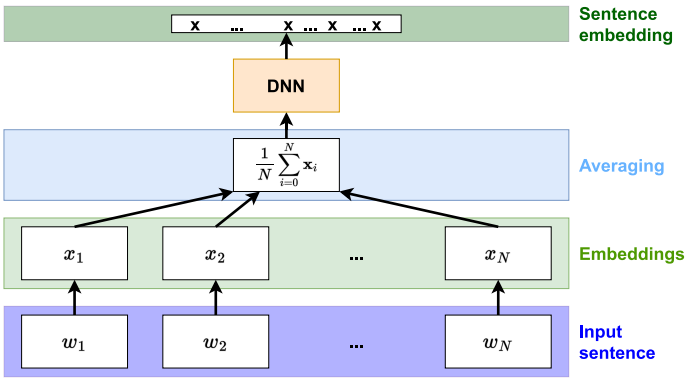


Figure 3.4: Architecture of the DAN model used for USE based on the textual description from [11]. The input words and bi-grams $(w_1, w_2, ..., w_N)$ are embedded. The embeddings are averaged and subsequently passed to a feedforward DNN, which produces a 512-dimensional sentence embedding.

The models are trained on both unsupervised training data, e.g., Wikipedia, and the supervised training dataset Stanford Natural Language Inference (SNLI) [9, 48].

The transformer model is more complex than the DAN model. More specifically, the transformer model complexity is $O(n^2)$, whereas the DAN model complexity is $O(n)$, n being the number of words in the sentence [9].

The memory usage of both models is equivalent to their complexity. Cer et al. state that DAN's memory usage is dominated by the parameters used to store the embeddings of the uni- and bi-grams. Moreover, the transformer model only stores the uni-gram embeddings and thus, can require less memory than DAN for short sentences [9].

3.3.4 InferSent

InferSent is a sentence embedding method trained in a supervised manner on the SNLI dataset [11, 48]. The trained model is transferable to other tasks.

Conneau et al. have compared multiple architectures in their work. The bi-directional Long Short-Term Memory (BiLSTM) architecture with max pooling was found to be the best option for the sentence encoder [11]. According to Reimers and Gurevych, it consists of a single siamese BiLSTM layer [48]. Given a sentence $(w_1, w_2, ..., w_T)$ of T words, the BiLSTM architecture computes the hidden representations h_t for each word w_t. The hidden representation h_t is the concatenation of the forward and backward hidden vectors $\overrightarrow{h_t}$ and $\overleftarrow{h_t}$. $\overrightarrow{h_t}$ and $\overleftarrow{h_t}$ are produced by a forward and backward Long Short-Term Memory (LSTM) respectively. Hence, the sentence is read from both directions and thus, considers past and future context. The architecture of the BiLSTM model with max pooling is depicted in Figure 3.5.

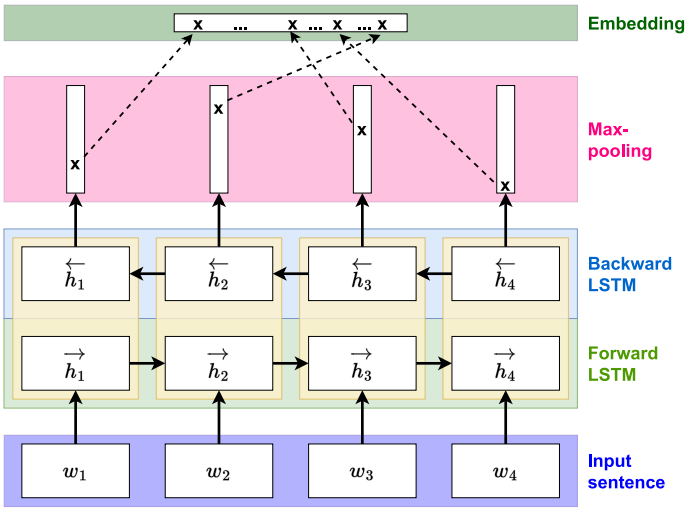


Figure 3.5: Architecture of the BiLSTM model with max pooling used for InferSent cf. [11]. The input sentence $(w_1, w_2, ..., w_T)$ is read from both directions by a forward and a backward LSTM producing $\overrightarrow{h_t}$ and $\overleftarrow{h_t}$ respectively. After concatenating $\overrightarrow{h_t}$ and $\overleftarrow{h_t}$ to h_t, max pooling is applied. The output is a fixed-sized embedding.

A LSTM is a Recurrent Neural Network (RNN) that is able to learn long-term dependencies. In other words: A LSTM is able to remember information as a so-called *state*. Certain LSTM mechanisms control whether the current state is deleted, whether new data is saved and to what degree the current state contributes to the current input processed in the node. Hence, LSTM nodes are not only influenced by the former output but also by their state.

Since the LSTM computes different numbers of hidden vectors h_t depending on the length of a sentence, a max pooling layer is applied to the hidden vectors. The max pooling layer selects the maximum value for each dimension of the hidden vectors.

3.3.5 Hugging face's SBERT

SBERT is an enhancement of Bidirectional Encoder Representations from Transformers (BERT). BERT is a pre-trained transformer network. It predicts a target value, for i.e. classification or regression tasks, based on two input sentences [48]. The input sentences are separated by a special token `[SEP]`. The base model applies multi-head attention over 12 transformer layers, whereas the large model applies multi-head attention over 24 transformer layers. The final label is derived from a regression function, which receives the output of the 12^{th} or 24^{th} layer, respectively. Reimers and Gurevych state that BERT is not suitable for specific pair regression tasks, since the number of input sentence combinations is too big. Another shortcoming of BERT is that it does not produce independent embeddings for single sentences. Moreover, Reimers and Gurevych found that common similarity measurements, for instance, the ones discussed in section 3.2, do not perform well on the representations of sentences in a VSM produced by BERT [48].

SBERT is a modification of BERT that provides fixed-sized embeddings for single sentences [48]. It differs from BERT in terms of architecture, since it adds a pooling layer after the BERT model. Reimers and Gurevych compare different pooling strategies, such as using the output of the `CLS` (i.e. first) token, mean pooling and max pooling. The architecture of a single SBERT network is depicted in Figure 3.6. In order to work with multiple input sentences at the same time, siamese and triplet network architecture, i.e. multiple BERT networks with tied weights, are constructed. To perform classification or inference tasks layers are added on top of the SBERT network. SBERT is trained on the SNLI dataset.

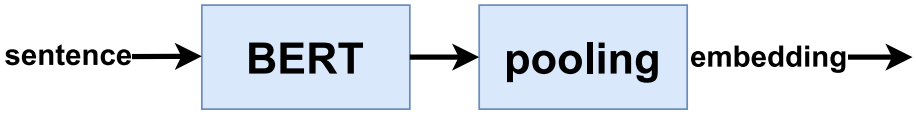
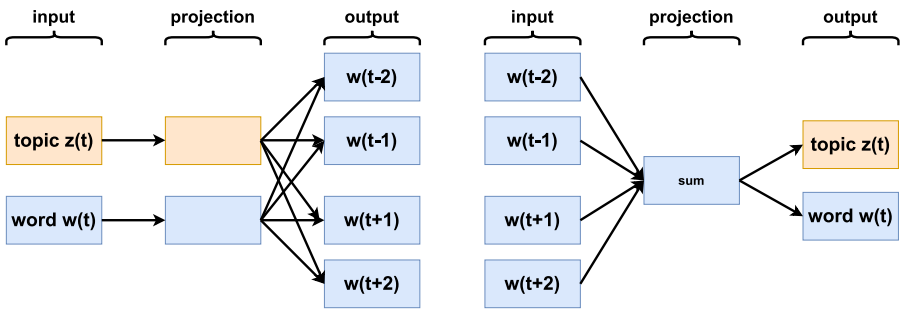


Figure 3.6: Architecture of SBERT cf. [48]. BERT is extended by a pooling layer. The input is a sentence and the output is a fixed-sized embedding.

According to Reimers and Gurevych, SBERT outperforms InferSent and USE.

3.4 Topic Modelling

Since more and more text data emerges, methods to analyse and extract information from texts become more important. One of these methods is topic modelling. A topic is defined as a cluster of words that frequently occur together. It can be described by a probability distribution over a vocabulary. A document can be represented by one or more topics. Common topic modelling algorithms are Latent Semantic Analysis (LSA), Probabilitstic Latent Semantic Analysis (PLSA), LDA and Correlated Topic Model (CTM) [3].



(a) Skip-gram architecture cf. [43]. (b) CBOW architecture cf. [43].

Figure 3.7: Both learning architectures of top2vec. $w(t-2), w(t-1), w(t+1), w(t+2)$ are the context words of the centre word $w(t)$ of topic $z(t)$.

3.4.1 BERTopic

3.4.2 LDA

LDA is a generative model, which recreates the original document word distributions with minimal error given the topics [3, 4]. A discrete multinomial probability distribution over a vocabulary consisting of W words is called a topic. A document is a mixture of K latent topics. Hence, each document from the document corpus D is represented by specific topic probabilities.

The probability distribution learnt by LDA only considers the statistical relationship of word occurrences in documents [43]. The content of a document can be described by the top N words with the highest conditional probability given a topic [43]. Due to the fact that high probability words are considered to be informative, LDA may regard words that occur frequently as important even though they might be uninformative in reality [4, 43].

3.4.3 Top2Vec

The approach top2vec was proprosed by Angelov [4]. It addresses several problems of state-of-the-art topic modelling approaches, such as LDA. Opposed to LDA, top2vec does not require the user to specify the number of topics k, i.e. it does not discretize the topic space into k topics, and it does not require stop word removal or lemmatization. Moreover, it considers the semantic meaning of words unlike LDA. In contrast to LDA, top2vec only associates one topic with a document.

top2vec is based on Word2Vec and Doc2Vec. The documents are embedded using PV-DBOW. There are two learning architectures adapted from Word2Vec to train the model, namely CBOW and Skip-gram [43]. They are depicted in Figure 3.7. The Skip-Gram learning

task is to predict the document a word came from [4, 43]. Similar to Doc2Vec, top2vec not only embeds words and documents in the same feature space but also topics [4, 43]. The similarity between embeddings can be measured using the cosine similarity function [43]. Angelov regards each point in the VSM as a topic, described by its nearest words.

Angelov states that topics are continuous and can be described by different sets of words [4]. Hence, topic modelling is defined as the task of finding sets of informative words that describe a document. Documents in dense areas of the topic space are considered to be about the same topic. The density-based clustering algorithm Hiercharical Density-Based Spatial Clustering of Applications with Noise (HDBSCAN) is used to find these dense areas. The topic vector is denoted as the centroid or average of the document vectors belonging to the topic. Angelov has compared several topic vector definitions and found the arithmetic mean to be the best option [4]. The number of topics is derived from the number of dense areas. It is possible to merge topics to hierarchically reduce the number of topics found to any number of topics smaller than the number initially found.

In order to find topics, clusters of documents, i.e. dense areas, need to be identified. The clustering algorithm HDBSCAN is used to find these dense areas. Since HDBSCAN has difficulties finding dense clusters in high-dimensional data, the dimensionality reduction method Uniform Manifold Approximation and Projection (UMAP) is applied [4]. The steps of the topic modelling procedure top2vec are depicted in Figure 3.8.

According to Niu and Dai, the complexity of top2vec is linear with the size of the dataset [43].

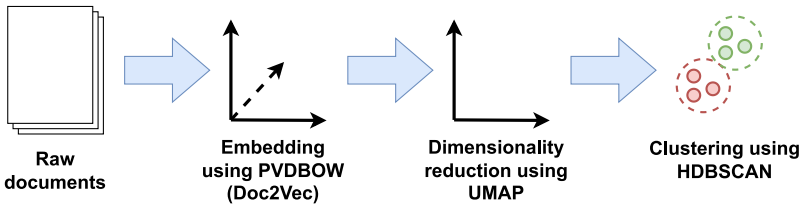


Figure 3.8: Procedure of topic modelling using top2vec.

3.4.4 Word Clouds

A Word Cloud is a technique to visualize the most predominant words in a text [31]. The size of a word correlates to its frequency or importance in the text. However, a word does not have to be meaningful to appear large. A Word Cloud does not provide information about the meaning or context of words and thus, one has to be careful when interpreting the results.

3.5 Compression of data

According to Radu et al., decomposition of data which preserves the inner structure in inherent clusters improves the quality of clusters obtained by cluster methods. Usually, the results of data analysis techniques improve when being applied to reasonably low dimensional data. Moreover, compressed data is less memory-consuming and often less difficult to interpret by humans since there are more methods to visualize low-dimensional data.

3.5.1 AE

The idea of this approach is to use a low-dimensional representation of the input, which is generated by an AE, to reduce complexity. The high-dimensional data is encoded into a low-dimensional representation using the encoder of an undercomplete AE [39]. The low-dimensional representation can be decoded into an approximation of the high-dimensional original using the decoder of the AE.

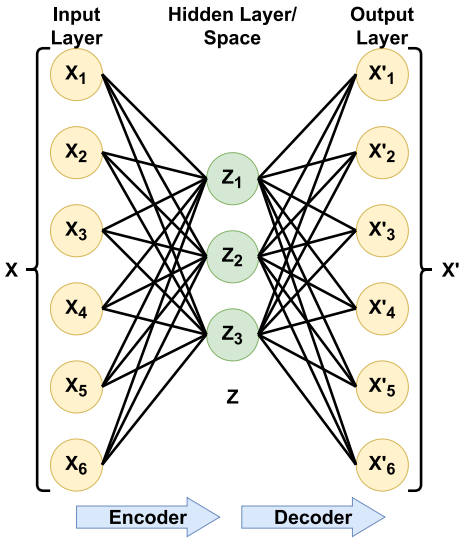


Figure 3.9: Structure of an AE cf. [39]

An undercomplete AE is a feed-forward NN, which consists of an encoder and a decoder. It learns efficient (non-correlated) encodings of the input data [39]. It is *undercomplete* because the dimensionality of the hidden layer, or so-called hidden space, is lower than the dimensionality of the input layer [29]. *Feed-forward* means that the information flows from the input layer to the output layer [29]. However, while training, the network employs backpropagation to update the parameters of the network [29].

The AE's goal is to approximate the identity function $f_\theta(X) = X$ (trivial solution eliminated) for input X and function parameters to be learned θ [29]. The input and output layers have the same dimensionality.

$$Z = f_E(W_\theta X + B_\theta) \tag{3.5}$$

The formulae for the encoder and the decoder are given in Equation 3.5 and in Equation 3.6. The parameter W_θ is the weight, whereas B_θ is the bias. The activation functions f_E and f_D are possibly non-linear and thus, the NN is capable of more than linear regression. Z is the low-dimensional representation of the input X and X' is the reconstructed version of Z.

$$X' = f_D(W_\theta Z + B_\theta) \tag{3.6}$$

The loss function L is defined as the reconstruction error between the input X and the output X' [29]. In order to train the AE, the loss function is minimized [34].

3.5.2 Eigenfaces

According to Turk and Pentland, the idea of Eigenfaces is inspired by information theory. Opposed to former approaches in the domain of face recognition which relied on the classification of images based on a set of predefined facial features, such as distance between eyes, Eigenfaces does not use predefined features [55]. The goal of this approach is to represent images using a smaller set of image features, i.e. compression to a lower-dimensional feature space, such that it is possible to distinguish between the images [55, 58]. These features do not necessarily correspond to human facial features [55]. Similar pictures, i.e. of the same person, should lie on a manifold in the lower-dimensional feature space [53]. The decomposition of input images not only reduces the complexity but also facilitates modelling probability density of a face image [53].

The greyscale input images are two-dimensional arrays of numbers: $\mathbf{x} = \{x_i, i \in \mathbf{S}\}$, $\mathbf{S}$ being a square lattice [61, 55]. The images are reshaped to an one-dimensional array $\mathbf{x} = [x_1, x_2, ..., x_n]^T \in \mathbb{R}^n$, where $n = \|\mathbf{S}\|$ and $\mathbb{R}^n$ is the n-dimensional euclidean space [61]. Some authors stress that the background is removed to omit values outside the face area [55]. In literature, typically, the original images' dimension is 512x512 [55]/ 64x64 [13], whereas the projected images' dimension is 16x16 [55]/ 250 [13]. Turk and Pentland stress that the data should be normalized, i.e. centred: $\Phi_k = \mathbf{x}_k - \psi$. Φ_k being the difference of the k-th training image and the average image $\psi = \frac{1}{N}\sum_{k=1}^{N} \mathbf{x}_k$, N being the number of training images.

The next step is to find an alternative lower-dimensional representation of the images, which preserves most of the information of the original image. In mathematical terms,

this decomposition can be expressed as $\mathbf{x} = \sum_{i=1}^{n} \hat{x}_i \mathbf{e}_i$, $\hat{x}_i$ being inner product of $\mathbf{x}$ and $\mathbf{e}_i$, $\mathbf{e}$ being an orthogonal basis [61]. If all basis vectors are used, the original image can be reconstructed using a linear combination of the basis vectors [55, 13]. The number of basis vectors is limited by the minimum of the training set size N [55] and the number of pixels n [13]. In order to compress the input from a $n-$ to a m-dimensional space, given $m \ll n$, only the first m basis vectors are used. The parameter m is chosen such that $\hat{x}_i$ is small for $i \geq m$ [61]. The compressed version of the image is denoted $\mathbf{x} \simeq \hat{\mathbf{x}} = [\hat{x}_1, \hat{x}_2, ..., \hat{x}_m]^T$. In other words: The compressed image is a vector of the first m weights of the linear combination of weight and basis vectors used to transform the image back to the original space [55]. The weights denote the position of the projection of the face images in the feature space or so-called face space spanned by the first m basis vectors [55].

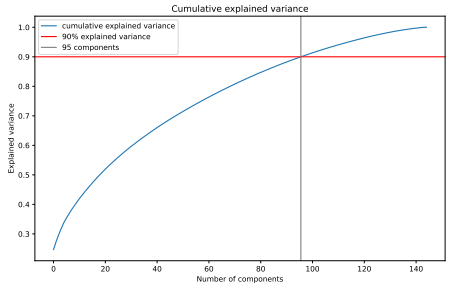
In the context of Eigenfaces one basis used for decomposition is the Karhonen-Loéve (KL) basis, i.e. PCA [61, 55]. According to Zhang et al., the KL representation is optimal in the sense that it minimizes the Root Mean Square Error (RMSE) between the original image and the compressed image calculated using $m < n$ orthogonal vectors. The KL basis consists of the eigenvectors of covariance matrix $\mathbf{C} = E\left[\mathbf{x}\mathbf{x}^T\right]$ of the input images $\mathbf{x}$ [61]. Since these eigenvectors can have facial features, they are called *Eigenfaces*. There are two approaches in the literature to determine the number of Eigenfaces m used to compress the input images:

(a) The cumulative explained variance of the first $i \leq n$ eigenvectors (sorted by eigenvalues λ_i) is calculated [61, 13, 52]. The eigenvalues λ_i can be interpreted as the amount of variance explained by the corresponding eigenvector $\mathbf{e}_i$, which is equivalent to information or entropy. The user can choose how much variance, i.e. information, should be preserved, by choosing m such that the explained variance is greater than a chosen threshold. Sudiana et al. use a threshold of 90%. A plot displaying the cumulative explained variance and a threshold of 90% is shown in Figure 3.10 (a).

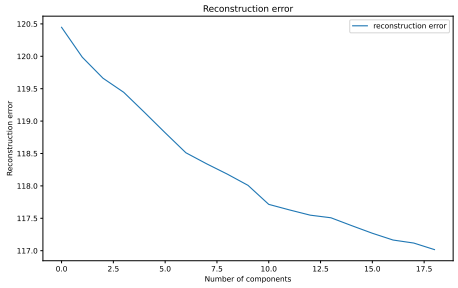
(b) The number of Eigenfaces m is chosen using the reconstruction error-complexity trade-off. The reconstruction error, i.e. the RMSE of the original image X and the inverse transformed image X' calculated in Equation 3.7 for different values of m. The "elbow" point marks the point where the reconstruction error decreases only slightly for increasing m and thus, is an indicator for the optimal m. A visualization of this approach is shown in Figure 3.10 (b).

$$\text{RSME} = \sqrt{\frac{\sum_{i=1}^{N}(x_i - x_i')^2}{N}} \tag{3.7}$$

In order to reduce calculation complexity, C is approximated. Zhang et al. propose the approximation $\mathbf{C} \simeq \frac{1}{N}\sum_{k=1}^{N} \mathbf{x}_k \mathbf{x}_k^T = \frac{1}{N}\mathbf{X}\mathbf{X}^T$, with $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_N]$, $\mathbf{x}_i \in \mathbb{R}^n$ [61].



(a) The cumulative explained variance of the first $i \leq n$ eigenvectors (sorted by eigenvalues λ_i).



(b) The reconstruction error RMSE calculated for different values of m. Around 13 is an "elbow" point.

Figure 3.10: Two approaches in the literature to determine the number of Eigenfaces m used to compress the input images.

Finding the eigenvectors of $\mathbf{X}\mathbf{X}^T$ is still computationally expensive, since $\mathbf{X}\mathbf{X}^T$ is a n by n matrix. According to Zhang et al., the eigenvectors of $\mathbf{X}\mathbf{X}^T$ can be calculated by using the eigenvectors of $\mathbf{X}^T\mathbf{X}$. The eigenvalues $\mathbf{e}_i \in \mathbb{R}^n$ of $\mathbf{X}\mathbf{X}^T$ can be derived from the eigenvectors $\mathbf{v}_i \in \mathbb{R}^N$ of $\mathbf{X}^T\mathbf{X}$ by $\mathbf{e}_i = \frac{1}{\sqrt{\lambda_i}}\mathbf{X}\mathbf{v}_i$ as discussed in more detail in [61]. Hence, the problem is reduced to a N by N matrix, which is computationally less expensive to solve, since $N \ll n$. Eigenvectors can be calculated using singular value decomposition (SVD) [61]. SVD is a method, which decomposes a matrix into the so-called left singular vector, the diagonal matrix and the right singular vector [6]. SVD facilitates the calculation of eigenvectors.

In the literature, face images are classified by comparing their position in the face space with those of already known faces [55]. According to [55], this approach performs well on datasets with little variation in pose, lighting and facial expression. However, Zhang et al. state, that the performance deteriorates if the variations increase since the changes introduce a bias that makes the distance function used to make classifications a no longer reliable measure.

3.6 Clustering

Clustering is used in a variety of domains to group data into meaningful subclasses, i.e. clusters [44, 14, 30]. According to Patwary et al. and Radu et al., common domains include anomaly detection, noise filtering, document clustering and image segmentation. The objective is to find clusters, which have a low inter-class similarity and a high intra-class similarity [44]. The similarity is measured by a distance function, which is dependent on the data type. Common distance functions are the Euclidean distance, the Manhattan distance and the Minkowski distance [30].

There are multiple clustering techniques, which can be divided into four categories [2]:

- **Hierarchical clustering**: Algorithms, that create spherical or convex-shaped clusters, possibly naturally occurring. A terminal condition has to be defined beforehand. Examples include CLINK, SLINK [14] and Ordering Points To Identify the Clustering Structure (OPTICS) [44].

- **Partitional based clustering**: Algorithms, that partition the data into k clusters, whereas k is given apriori. Clusters are shaped in a spherical manner, are similar in size and not necessarily naturally occurring. KMeans is a popular example of a partitional-based clustering algorithm.

- **Density based clustering**: Density is defined as the number of objects within a certain distance of each other [30]. The resulting clusters can be of arbitrary shape and size. The algorithm usually chooses the optimal number of clusters given the input data. However, some algorithms are sensitive to input parameters, such as radius, minimum number of points and threshold. Popular examples are Density-Based Spatial Clustering of Applications with Noise (DBSCAN) and OPTICS.

- **Grid based clustering**: Similar to density-based clustering, but according to Agrawal et al. better than density-based clustering. Examples include flexible grid-based clustering [14].

Multiple approaches listed below use the term *ε-neighbourhood*, which is defined as the set of all objects within a certain distance ε of a given object [44]. In other words: $N_\varepsilon(x) = \{y \in X | dist(x, y) \leq \varepsilon, y \neq x\}$, ε being the so-called generating distance.

3.6.1 KMeans

KMeans partitions the data into $k \in \mathbb{N}$ clusters, k is given apriori [30, 46]. First, k centroids, i.e. cluster centres, are randomly initialized. Then, the objects are assigned to the closest centroid. Afterwards, the centroids are updated by calculating the mean of the assigned objects. The process is repeated until the terminating condition, for instance, no more change in the clusters, is met [30]. By iteratively reassinging the objects to the closest centroid and updating the centroids, the algorithm minimizes the within-cluster sum of squared errors E, i.e. the sum of squared (Euclidean) distances between objects in a cluster and their centroid μ_i, calculated in Equation 3.8 from [30], where C_i is the i-th cluster.

$$E = \sum_{i=1}^{k} \sum_{x \in C_i} \|x - \mu_i\|^2 \tag{3.8}$$

Kanagala and Krishnaiah claim, that KMeans does not identify outliers.

3.6.2 DBSCAN

The clusters identified by DBSCAN have a high density and are separated by low-density regions [30]. In order to create clusters of minimum size and density, DBSCAN distinguishes between three types of objects [30]:

- **Core objects**: An object x with at least $minPts \in \mathbb{N}$ objects in its ε-neighbourhood $N_\varepsilon(x)$, i.e. $|N_\varepsilon(x)| \geq minPts$ is true [44].
- **Border objects**: An object with less than $minPts$ objects in its ε-neighbourhood, which is in the ε-neighbourhood of a core object.
- **Noise objects**: An object, which is neither a core object nor a border object.

Kanagala and Krishnaiah define $y \in X$ as *directly density reachable* from $x \in X$, if y is in the ε-neighbourhood of core object x [30]. Moreover, a point $y \in X$ is *density reachable* from $x \in X$, if there is a chain of objects $x_1, ..., x_n$ with $x_1 = x$ and $x_n = y$, which are directly density reachable from each other as displayed in Figure 3.11 [30].

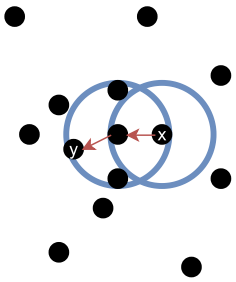


Figure 3.11: Density reachability cf. [5]. The point $y \in X$ is density reachable from $x \in X$, since there is a chain of directly density reachable objects x, o, y.

The points $x \in X$ and $y \in X$ are said to be *density connected*, if there is an object o, from which both x and y are density reachable [30]. Density connectivity is visualized in Figure 3.12.

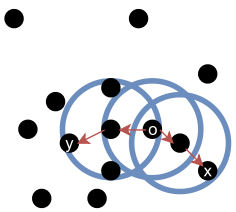


Figure 3.12: Density connectivity cf. [5]. The objects x and y are density connected since there is an object o, from which both x and y are density reachable.

The DBSCAN algorithm starts by labelling all objects as core, border or noise points. Then, it eliminates noise points and links all core points, which are within each other's neighbourhood [30]. Groups of connected core points form a cluster [30]. In the end, every border point is assigned to a cluster [30]. The non-core point cluster assigning is non-deterministic [44]. This algorithm creates clusters as a maximal set of density-connected points [30].

According to Kanagala and Krishnaiah, DBSCAN can identify outliers or noise. However, the algorithm is sensitive to the input parameters $minPts$ and ε and has difficulties distinguishing closely located clusters [30]. Moreover, if one wants to obtain hierarchical clustering, one has to run the algorithm multiple times with different ε, which is expensive in terms of memory usage [44]. According to Radu et al., DBSCAN is affected by the curse of dimensionality. Since DBSCAN relies on nearest neighbour queries and these become less meaningful in high dimensions, i.e. distances become difficult to interprete, the quality and accuracy of the results declines with increasing dimensionality [46]. Radu et al. found that their DBSCAN model assigned most documents noise, when the dimensionality was sufficiently large.

3.6.3 OPTICS

OPTICS does not return an explicit clustering, but rather a density-based clustering structure of the data, which is equivalent to clustering results of a broad range of parameters [5]. Ankerst et al. claim that real-world datasets cannot be described by a single global density, since they often consist of different local densities, as displayed in Figure 3.13.

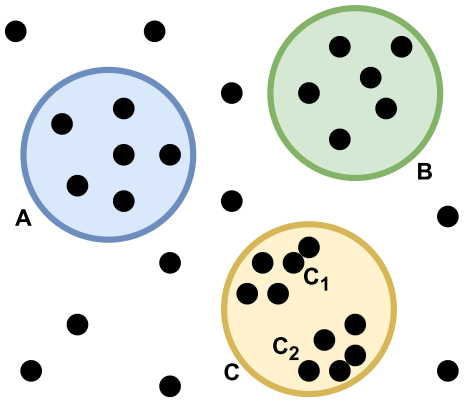


Figure 3.13: Clusters with different densities cf. [5]. Since C_1 and C_2 have different densities than A and B, a clustering algorithm using one global density parameter would detect the clusters A, B and C, rather than A, B, C_1 and C_2 .

Opposed to DBSCAN, OPTICS is able to detect clusters of varying densities [14]. OPTICS produces an order of the elements according to the distance to the already added elements [14, 44]: The first element added to the order list is arbitrary. The order list is iteratively expanded by adding the element of the ε-neighbourhood to the order list, which has the

smallest distance to any of the elements already in the order list. Hence, clusters with higher density, i.e. lower ε, are added first (prioritized) [30, 5]. When there are no more elements in the ε-neighbourhood to add, the process is repeated for the other clusters. The non-core point cluster assigning is non-deterministic [44].

$$RD(y) = \begin{cases} \text{NULL} & \text{if } |N_\varepsilon(x)| < minPts \\ max(core\_dist(x), dist(x,y)) & \text{otherwise} \end{cases} \tag{3.9}$$

OPTICS saves the reachability distance $RD(y)$, as calculated in Equation 3.9 from [44], with core distance $core\_dist$ being the minimal distance ε^{min} such that $|N_{\varepsilon^{min}}(x)| \geq minPts$ (i.e. the distance to the $minPts^{th}$ point in N_ε) or NULL else, of each element y to its predecessor x in the order list and thus, a representation of the density necessary to keep two consecutive objects x and y in the same cluster [44]. If $\varepsilon < RD(y)$, then y is not density reachable from any of its predecessors and thus, one can determine whether two points are in the same cluster for given information saved by OPTICS [44, 5]. If the core distance of an element is not NULL, i.e. it is a core object, and it is not density reachable from its predecessors, it is the start of a new cluster [5]. Otherwise, the element is a noise point [5]. According to Patwary et al., the algorithm builds a spanning tree, which enables obtaining the clusters for a given ε by returning the connected components of the spanning tree after omitting all edges with $\varepsilon < RD(y)$ [44]. The relationship between ε, cluster density and nested density-based clusters is displayed in Figure 3.14.

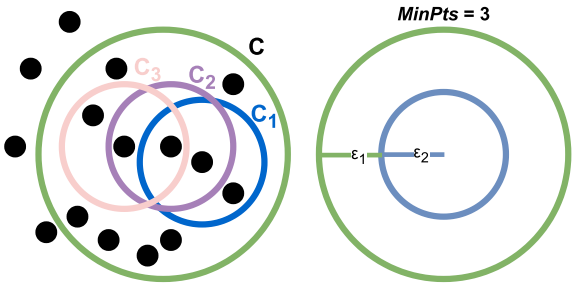


Figure 3.14: The relationship between ε, cluster density and nested density-based clusters cf. [5]. For a constant $minPts$, clusters with higher density such as C_1, C_2 and C_3, i.e. a low ε_2 value, are completely contained in lower density clusters such as C given $\varepsilon_1 > \varepsilon_2$. This idea forms the basis of OPTICS of expanding clusters iteratively and thus, enables the detection of clusters for a broad range of neighbourhood radii $0 \leq \varepsilon_i \leq \varepsilon$.

This procedure enables the extraction of clusters for arbitrary $0 \leq \varepsilon_i \leq \varepsilon$ [30, 5]. According to Patwary et al.'s work, even though the clustering algorithm is expensive the extraction only needs linear time. According to [5], the algorithm yields good results if the input parameters $minPts$ and ε are "large enough" and thus, the algorithm is rather insensitive to the input parameters.

The smaller ε is chosen, the more objects will be identified as noise and thus, the algorithm will not identify clusters with low density, since some objects only become core objects for a larger ε [5]. According to Ankerst et al., the optimal value for ε creates one cluster for most of the objects with respect to a constant $minPts$, since information about all density-based clusters for $\varepsilon_i < \varepsilon$ is preserved. A heuristic for choosing ε based on the expected k-nearest neighbour distance is presented in [5].

High values for $minPts$ smoothen the reachability curve, even though the overall shape stays roughly the same [5]. According to Ankerst et al., the optimal value for $minPts$ is between 10 and 20.

3.7 Database Elasticsearch

Elasticsearch is a widely used non-relational database, which was designed to store and perform full-text search on a large corpus of unstructured data [57]. This open-source distributed document-driven database system is built in Java and is based on the Apache Lucene (Java) library for high-speed full-text search [57, 60]. According to Zamfir et al., Elasticsearch provides Wikipedia's full-text search and suggestions as well as Github's code search and Stack Overflow's geolocation queries and related questions. It enables near real-time search by index refreshing periods of one second. Needless to say, Elasticsearch is qualified to handle Big Data.

Elasticsearch is a document store, which stores schemaless key-value pairs called documents [25]. Elasticsearch's entries, i.e. documents, are stored in logical units, so-called indices. As stated by Zamfir et al. and Voit et al., the indices are structured similarly to Apache Lucene's inverted index format. An index can be spread into multiple nodes. A node is a single running instance of Elasticsearch [60]. An index is divided into one or more shards, which can be stored on different servers and enable parallelization [60].

Elasticsearch indices' entries are documents, which are saved in a JavaScript Object Notation (JSON) format [57]. A document's fields and field types are defined by the user when initializing the database index. By default, every field of a document is indexed and searchable [60].

Replicas are copies of shards, which create redundancy and thus, ensure availability [60].

By specifying the unique `_id` of a document and the database `index`, it is possible to retrieve a specific document from the database using the `GET Application Programming Interface (API)`. The query is real-time by default. The parameters `_source_excludes` or `_source_includes` may be used to exclude or include specific fields of the document in the response [17].

The keyword used when performing a full-text search is `match`. To query for a specific value, one has to specify the `<field>` of interest and the query value.

Elasticsearch preprocesses the query value before starting the search [23]. The default preprocessing steps of the so-called default analyzer include tokenization and lowercasing [23]. Omitting stop words is disabled by default, but it is possible to provide custom stop words or use the English stop word list [23]. It is possible to create custom tokenizers, which split the query value into tokens of a certain maximum length.

Another useful feature of Elasticsearch is the multi-term synonym expansion. When the user queries a specific phrase Elasticsearch expands the query to include synonyms of the query terms [22]. The maximum number of expansion terms is set to 50 by default but can be configured by the user [21]. By default, the multi-terms synonym expansion option is enabled [21].

Elasticsearch also provides the option to perform fuzzy matching instead of exact search. By enabling the fuzzy matching option, a Elasticsearch query consisting of for instance, *Bahama* returns documents that contain the word *Bahamas*. By default, this option is not enabled but can be enabled and configured individually by the user [21].

Another search option of Elasticsearch is the k-nearest neighbor (kNN) search. The return value of a kNN search is the `k` nearest neighbors in terms of a certain distance function of a query vector [35]. The query is a dense vector of the same dimension as the vectors stored in the database. According to [18], the kNN either returns the exact brute-force nearest neighbors or an approximation of the nearest neighbors calculated by the Hiercharical Navigable Small World (HNSW) algorithm [35, 18]. HNSW is a graph-based algorithm [35]. The term `navigable` refers to the graphs used, which are graphs with (poly-)logarithmic scaling of links traversed during greedy traversal concerning the network size [35]. The idea of a `hiercharical` algorithm is to create a multilayer graph, grouping links according to their link length, as displayed in Figure 3.15. The search starts on the uppermost layer, i.e. the layer containing the longest links, greedily traversing the layer until reaching the local minimum. It uses this local minimum as the starting point at the next lower layer and the process is repeated until the lowest layer is reached [35]. The layers of the graph are built incrementally, and a neighbour selection heuristic, as depicted in Figure 3.16, not only creates links between close elements, but also between isolated clusters to ensure global connectivity [35].

In order to perform the kNN search on a `<field>` it has to be of type `dense_vector`, indexed and a `similarity` measure has to be defined when initializing the database [18].

Elasticsearch's kNN implementation not only allows literal matching on search terms but also semantic search [18].

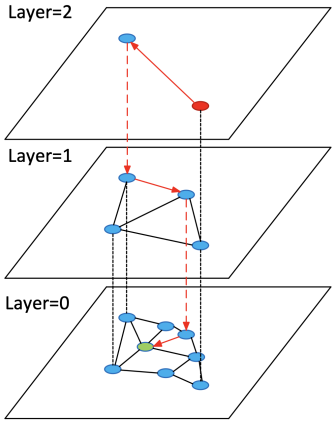


Figure 3.15: Structure of HNSW layers from [35]. The search starts on the uppermost layer, i.e. the layer containing the longest links, greedily traversing the layer until reaching the local minimum. The local minimum is used as the starting point at the next lower layer and the process is repeated until the lowest layer is reached.

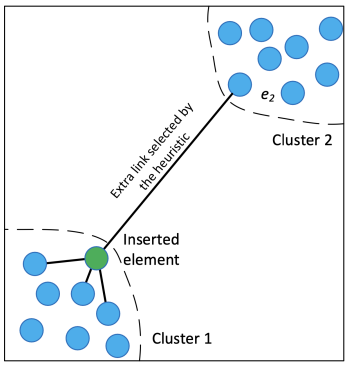


Figure 3.16: Neighbour selection heuristic of HNSW from [35]. The heuristic creates diverse links, i.e. links between close elements (e.g., green circle and elements in cluster 1) and between isolated clusters (e.g., green circle and e_2) to ensure global connectivity.

Besides Elasticsearch, the elastic stack offers other tools, for instance, Kibana, which provides a user interface to manage different models. After saving a model in Kibana, it is possible to create a text embedding ingest pipeline, which embeds new documents or reindexes existing documents [19].

3.8 Flask

Flask is open source and written in Python by Armin Ronancher in 2004 [7, 41]. According to Copperwaite and Leifer and Mufid et al., Flask is one of the most popular Python web frameworks. It provides powerful libraries for core functionality such as routing, templating, and Hypertext Transfer Protocol (HTTP) request parsing [12]. It is extensible and thus, can be extended with additional plugins without affecting the internal structure of the existing system [7].

Flask uses the Jinja Template Engine for template files including Hypertext Markup Language (HTML) pages, whereas static files such as Cascading Style Sheet (CSS) files are handeled using the Werkzeug WSGI toolkit [7]. According to Aslam et al., Jinja is modeled after the Django template system. Werkzeug implements, for instance, requests and response objects [41].

All requests received from clients are passed to an instance of the Flask application [28]. Hence, the first step is to create an instance of the Flask class, such as done in Listing 3.1.

```
1 app = Flask(__name__)
```

Listing 3.1: Initialization of Flask application instance.

Clients send requests to the web server, which passes them to the Flask application instance. The queries are then routed to the corresponding functions. Routing is the process of mapping Uniform Resource Locator (URL) paths to functions [28]. To define a route, the `route` decorator is used as displayed in Listing 3.2.

```
1 @api.route('/documents/<id>', endpoint='document')
2 class Document(Resource):
3     def get(self, id):
4         elastic_search_client = Elasticsearch(CLIENT_ADDR)
5         return query_database.get_doc_meta_data(elastic_search_client,
6             doc_id=id)
```

Listing 3.2: Exemplartary definition of a function to display routing with Flask. The `route` decorator is used to define the URL path.

URL can contain dynamic components, which are enclosed in `<>` angle brackets. The values of these components are passed to the function as arguments [28]. By default, dynamic components are of type `string`. However, other types including `int` and `float` are supported [28].

During development, the Flask application can be run using `flask run` to start the built-in development web server [28]. By enabling debug mode, the server automatically reloads the application when changes are detected [28].

An endpoint is a class with certain methods, which can be accessed using HTTP requests. Every endpoint can have multiple decorators, including `GET`, `POST`, `PUT` and `DELETE` [25]. The `GET` method is used to retrieve data from the server, whereas the other methods are used to either insert, update or delete data.

3.9 Angular

Angular is a framework for building web applications. It uses Node.js and TypeScript. Usually, the source code is structured into different modules, including components and services. Components are used to define the appearance of the application, while services are used to define the logic of the application and communicate with the backend.

Angular applications are created using the `ng new NAME` command line interface [49]. This command creates a skeleton, which can be customized to meet the needs of the application. By running `ng serve` the application can be served locally.

4 Implementation

This chapter describes how the theoretical basics from chapter 3 interplay and how they are used in this thesis. Specific parameter choices are explained in chapter 5.

4.1 Slurm

Since the data corpus is too big to be processed locally on a Apple M2 Pro MNW83D/A with 16 GB RAM and 12 cores, the Chair Intelligent Embedded Systems has offered to provide computally means to solve this problem. The scripts can be processed by multiple nodes which are managed by Slurm.

Slurm is an open-source management tool for Linux clusters [1]. It allocates resources, i.e. compute nodes, and provides the means to start, execute and monitor jobs [1, 59].

The so-called Slurm daemons control nodes, partitions, jobs and job steps [1]. According to slu, a partition is a group of nodes and a job is the allocation of resources, i.e. compute nodes, to a user for a limited period of time. A basic visualization of the architecture is given in Figure 4.1.

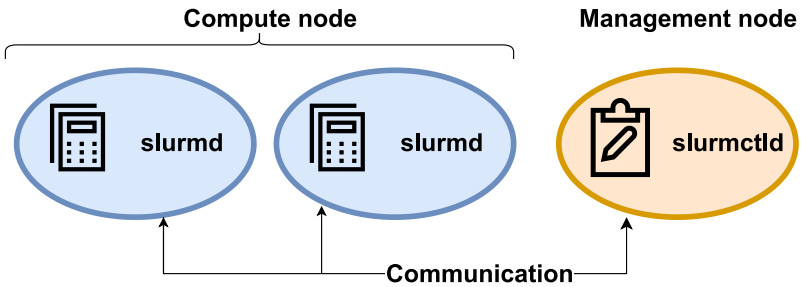


Figure 4.1: Slurm architecture. The management node has a `slurmctld` daemon, while every compute node has a `slurmd` daemon. The nodes communicate. The user can use certain commands, for instance `srun` and `squeue`, anywhere on the cluster.

Table 4.1: Fields of the Elasticsearch database index *Bahamas*.

Field name	**Field description**
_id	Unique identifier of document `i`. The identifier is generated by the sha256 hash algorithm from hashlib using the PDF file as input.
doc2vec	55 dimensional doc2vec embedding of `i`.
sim_docs_tfidf	sim_docs_tfidf embedding + all-zero flag of `i`. The all-zero flag is one if the TF-IDF embedding consists of only zeros, zero else. If the embedding's dimensionality is greater than 2048, the encoder of a trained AE is used to compress the embedding.
google_univ_sent_encoding	512 dimensional google_univ_sent_encoding embedding of `i`.
huggingface_sent_transformer	384 dimensional huggingface_sent_transformer embedding of `i`.
inferSent_AE	inferSent_AE embedding of `i`. Since the pretrained InferSent model embedding's dimension is 4096, the encoder of a trained AE is added to reduce the dimension to 2048.
pca_image	13-dimensional PCA version of first page image of `i`.
pca_optics_cluster	Cluster of `i` identified by OPTICS on PCA version of image.
argmax_pca_cluster	Number of maximum PCA component as cluster of `i`.
text	Text of `i`.
path	Path to `i`.

4.2 Elasticsearch

First, the content of the database is described, then, the initialization, insertion and updating process of filling the database are explained and finally, the process of querying is outlined.

Content of the database

In this work, the database is filled once with data from a large unstructured corpus of Portable Document Format (PDF) files. After the initialization of the database, it is used for queries. Therefore, the workflow is completely offline.

The index *Bahamas* stores different embeddings of the information derived from the text layer and metadata of the documents. As depicted in Figure 4.2, not only textual information is stored in the database, but also information about the appearance of the first page of the PDF. The structure of the index is presented in Table 4.1.

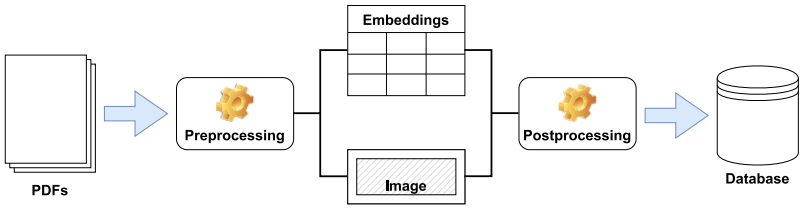


Figure 4.2: PDFs to Database. First, the data is preprocessed: The first page of a PDF file is converted to an image and the complete text is extracted. The images are stored in the database as well as the text and different embeddings of the text. Some values, such as the image or the InferSent embedding, have to be compressed to become an at most 2048 valued vector.

Initialization, insertion and updating

To facilitate working with and running the code the initialization of the database is split into multiple steps. As depicted in Figure 4.3, first the database is initialized by defining the index name and the mappings, i.e. the field names, types and sizes. This step is carried out using the keyword `create`.

Figure 4.3: Procedure of initialization and filling of the database.

Afterwards, the documents are created. The initial creation of a document only defines the fields `id`, `text` and `path`. In order to maximise efficiency when updating the database, the Elasticsearch's built-in functionality `bulk` is used. `bulk` sends chunks of multiple requests to the database. As displayed in Listing 4.1, `bulk` is called with the Elasticsearch client, a function which yields requests and parameters, which define the return values. The structure of a function that yields requests is shown in Listing 4.2.

```
1 bulk(client, create_document_aux(src_paths, client), stats_only= True)
```

Listing 4.1: Usage of Elasticsearch's helper functionality `bulk` to send multiple requests to the database in chunks.

The embeddings are added to the documents in a third step. To make sure that it is possible to update embeddings individually without changing other fields, a method to insert the embeddings of a specific model for all documents is created. The documents are updated using the `update` keyword and `bulk`.

```python
def create_document_aux(src_paths: list, client: Elasticsearch):
    for path in src_paths:
        id = get_hash_file(path)
        if get_doc_meta_data(client, doc_id=id) is not None:
            continue
        text = pdf_to_str(path)
        yield { '_op_type': 'create',
                '_index': 'bahamas',
                '_id': id,
                "text": text,
                "path": path}
```

Listing 4.2: Method that yields requests for `bulk`. The method checks if the document is already in the database and if not, it yields a request to create the document.

Queries

The default analyzer is used for the full-text search since for instance configuring a maximum token length did not seem necessary or likely to improve the results.

```python
results = elastic_search_client.search(
    index='bahamas',
    size=count,
    from_=(page*count),
    query= {'match' : {
                'text': {   'query':text,
                            'fuzziness': 'AUTO',}
            },
        }, source_includes=SRC_INCLUDES)
```

Listing 4.3: Exemplartary query to an Elasticsearchdatabase index. The number of results to return `size` and the start index of the results `from_` is defined. To enable fuzzy search a value for `fuzziness` has to be defined.

Moreover, the fuzzy matching option is set to `AUTO`, which means in terms of keyword or text fields that the allowed Levenshtein Edit Distance, i.e. number of characters changed to create an exact match between two terms, to be considered a match, is correlated to the length of the term [16]. By default, terms of length up to two characters must match exactly, terms of length three to five characters must have an edit distance of one and terms of length six or more characters must have an edit distance of two [16]. An exemplary query, which uses fuzzy search is given in Listing 4.3.

According to Malkov and Yashunin, one of kNN search's use cases is semantic document retrieval, which makes it a good fit for this task. In this work, the approximate nearest neighbours search is used, since it is faster and the results are good enough for the use

case of this work. The similarity measure used in this work is the cosine similarity, which calculates the `_score` of a document according to Equation 4.1 from [20], where `query` is the query vector and `vector` is the vector representation of the document in the database. The other similarity measures provided by Elasticsearch are `l2_norm` or so-called Euclidian distance and `dot_product` which is the non-auto-normalized version of the `cosine` option. Since cosine is not defined on vectors with zero magnitude, embeddings that can return all zero vector representations, such as sim_docs_tfidf, are enhanced with an all-zero flag before inserting them into the database.

$$\frac{1 + \text{cosine}(\text{query}, \text{vector})}{2} \tag{4.1}$$

In this work, the only tool from the elastic stack used is Elasticsearch. Without Kibana, the used models are saved on disk as Pickle (PKL) files. Consequently, instead of using the kNN query structure for semantic search on embeddings provided by Elasticsearch, the normal kNN search on a field that contains an embedding is used.

4.3 Eigendocs

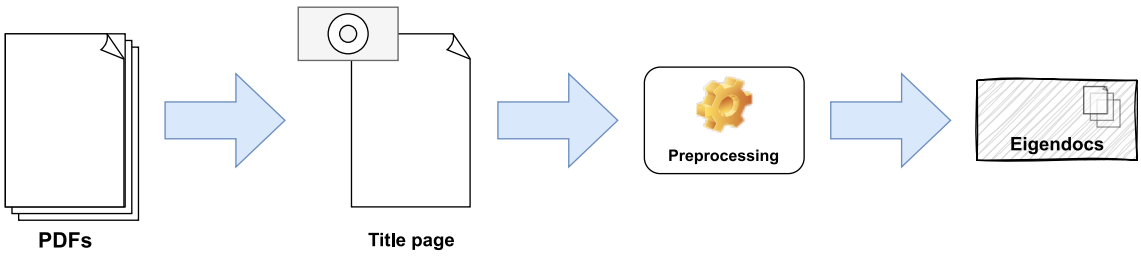


Figure 4.4: From PDFs to Eigendocs. Firstly, the first page of a document is converted to an image. Then, the image is preprocessed: It is placed on a white canvas, to ensure all images have the same dimensions. Moreover, it is converted to greyscale and normalized to values between zero and one. Afterwards, the 2d image is reshaped to a 1d array. Lastly, the image is compressed using Eigendocs.

In this work, the Eigenfaces approach from subsection 3.5.2 is used to compress the images of the first page of documents. The idea is that documents not only hold textual information but also visual information, such as layout, company logo or signature. By mapping those images on a subspace, they ought to be grouped by visual similarity. The procedure of the Eigenfaces adaption *Eigendocs* is displayed in Figure 4.4.

The documents are first read from a directory. Subsequently, their first page is converted to an image and saved. When initially filling the database, these images are read from their directory. Firstly, the maximum height and width among all images in the corpus is calculated. These dimensions are used to create a white canvas for each image which

forms the background. Every image is placed in the upper left corner. Hence, assuming the selection of documents used to fit the PCA model is representative, scaling is not necessary and thus, the portion of white pixels on the right and bottom side encodes the dimension of the former image. However, some images are bigger than the max values from selected data and as a consequence are scaled. Therefore, the relative size of images in the corpus is incorporated in the resulting representation of the input images.

```
C = np.ones((max_w,max_h))
C[:doc.shape[0],:doc.shape[1]] = rgb2gray(doc)
documents.append(C.ravel())
```

Listing 4.4: Preprocessing of the input images from Dr. Christian Gruhl. The background is a white canvas. The images are converted to one-dimensional greyscale values.

Afterwards, they are converted to greyscale images using Listing 4.5. Before returning the image, the two-dimensional image vectors are converted to one-dimensional ones as displayed in the last line of Listing 4.4. The decomposition is transformed using PCA as displayed in Listing 4.6. The implementation of PCA from sklearn intrinsically normalizes the data as described in subsection 3.5.2 and thus, does not require the user to manually preprocess the data.

```
0.299*img[:,:,0] + 0.587*img[:,:,1] + 0.114*img[:,:,2]
```

Listing 4.5: Conversion of RGB pixel values to greyscale from a script by Dr. Christian Gruhl.

```
pca = decomposition.PCA(n_components=n_components, whiten=True,
    svd_solver="randomized")
```

Listing 4.6: Initialization of the PCA instace used to compress the image data. Since the Eigenfaces approach uses a svd_solver, the adaption Eigendocs has to be implemented likewise.

4.4 Autoencoder

In this work, the AE is used to reduce the dimensionality of the InferSent and the TF-IDF embedding. Since the InferSent model is pretrained, it is not possible to change the dimensionality of the embedding without a considerably big effort, i.e. retraining the model on a sufficiently large data corpus and reconfiguring the model's parameters. Moreover, retraining the model would destroy the purpose of its presence in this work, which is to provide a pretrained model and thus, reducing the complexity of training a model. Therefore, it is not feasible to change the dimensionality of the InferSent embedding, but rather adding a

supplementary layer after the model produce the final embedding. Similarly, the TF-IDF embedding correlates with the vocabulary size and thus, the size od the data corpus. Further reducing the vocabulary size would decrease the TF-IDF model's quality. Hence, the idea is to use the encoder of an AE to reduce the dimensionality of the InferSent and the TF-IDF embedding.

The implementation was provided by a blog post using keras and is adapted to fulfil the needs of the specific context. The number of hidden layers and their dimensionality were increased to improve the ability of the model to reconstruct.

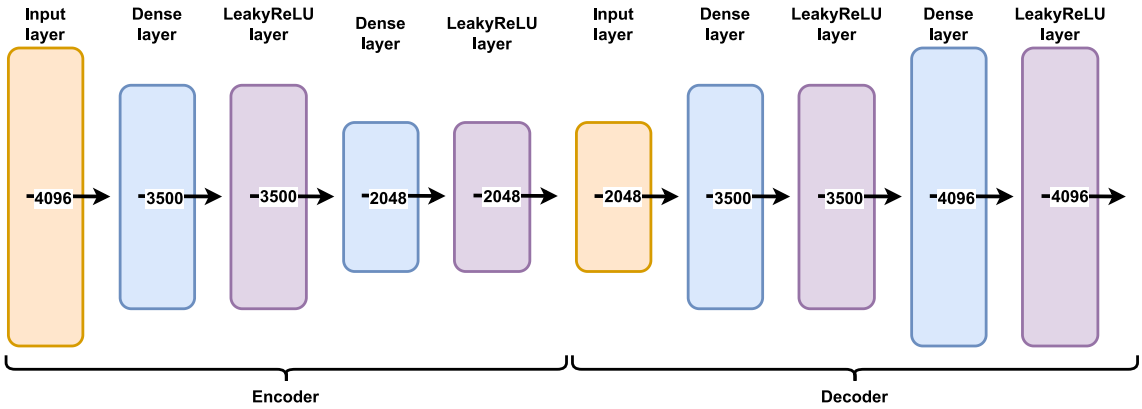


Figure 4.5: Architecture of the AE

The architecture of the AE is presented in Figure 4.5. In this work, only the encoder is used.

4.5 TF-IDF

The TF-IDF model has to be initialized and trained on the data corpus to build the data-specific vocabulary. An exemplary implementation is given in Listing 4.7. The `TfidfVectorizer` is provided by the `scikit-learn` package. When initializing the model, the parameters define not only the input type but also the way the data is preprocessed. The `input` parameter defines the input type, i.e. `content` means that the input is a list of strings or bytes, whereas `file` assumes the input has a `read` method and `filename` denotes a list of filenames as input [**?**].

The `preprocessor` parameter defines the preprocessing (string transformation) stage. It is possible to override the default with a custom preprocessing function. The parameters `min_df` and `max_df` define the minimum and maximum document frequency of a word in the corpus. The default values are 1, i.e. a term has to appear at least once, and 1.0, i.e. a term appears at most in all documents, respectively [**?**].

By default, the `scikit-learn` implementation uses the `norm='l2'` parameter, i.e. the Euclidean norm [54]. The implementation of TF-IDF in `scikit-learn` is different from the original TF-IDF definition. The difference is the calculation of the IDF part, which is given in Equation 4.2 from [54]. The one is added to M_{ij} due to the parameter `smooth_id=True` by default to prevent zero divisions [54] and to avoid logarithmic divergences due to a zero argument [45]. After calculating the TF-IDF values, they are normalized by the Euclidean norm $v_{norm} = \frac{v}{\|v\|_2} = \frac{v}{\sqrt{v_1^2+v_2^2+...+v_M^2}}$.

$$\text{idf}(w_{ij}) = \log \frac{1+M}{1+M_{ij}} + 1 \tag{4.2}$$

```
tfidf = TfidfVectorizer(input='content',
            preprocessor=TfidfTextPreprocessor().transform, min_df=3,
            max_df=int(len(docs)*0.07))
tfidf.fit(documents)
```

Listing 4.7: Initialization of the TF-IDF model. Firstly, an instance of the `TfidfVectorizer` class is created. Secondly, the `fit` method is called to fit the model on the documents.

In this work, the text of the PDFs is first extracted, then preprocessed using a custom preprocessor and afterwards embedded using the `TfidfVectorizer`, which returns the TF-IDF weights as embedding. Before storing the TF-IDF weights in the database, they are enhanced with an all-zero flag. The all-zero flag ensures that no all-zero vectors are stored in the database by extending those that have a zero magnitude with a "1" entry and "0" otherwise. All-zero TF-IDF weights indicate that a document does not have any terms with the vocabulary in common. Since the vocabulary is kept relatively small with respect to the number of different words in the data corpus to reduce the dimensionality of the embeddings, it is not unlikely that a document does not contain any of the vocabulary terms. The all-zero flag is necessary because the cosine similarity used to query for similar documents in the database cannot handle vectors of zero magnitude. The pipeline in Figure 4.6 visualizes the steps.

In this work, a custom preprocessing function is used. The preprocessing steps are visualized in Figure 4.7 on an exemplary text. Firstly, the accents are stripped from the text. Then, all new line symbols are replaced with a whitespace. Afterwards, the text is converted to lowercase. Then the numbers are discretized, i.e. all numbers between 0 and 99999 are replaced with the string `SMALLNUMBER`, numbers bigger than 99999 are replaced with the string `BIGNUMBER` and floats are replaced with the string `FLOAT`. The next step is to remove all punctuation symbols. To ensure eventual empty tokens generated by the preprocessing steps stated above are omitted, all sequences of multiple subsequent whitespaces are discarded. After that, the symbols for numbers are enclosed with pointed brackets, e.g. `<SMALLNUMBER>`. Then, the text is tokenized, i.e. splitting at whitespaces, and stop words

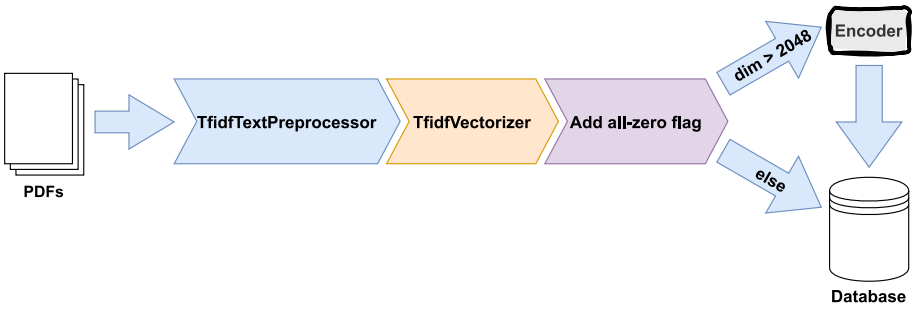


Figure 4.6: TF-IDF pipeline. Firstly, the text extracted from the documents is preprocessed using a custom preprocessor. Then, the TF-IDF values are obtained from the `TfidfVectorizer`. Afterwards, the all-zero flag is added to the TF-IDF weights. If the resulting dimensionality is bigger than 2048, the encoder of an AE is used to reduce the dimensionality. The results are stored in the database.

are omitted. The `nltk` package provides the stop words used to omit uninformative words. The stop word corpus consists of common English stop words. Afterwards, the tokens are lemmatized. The lemmatizer used is `WordNetLemmatizer` from the `nltk` package. In the end, the tokens are joined to a string again.

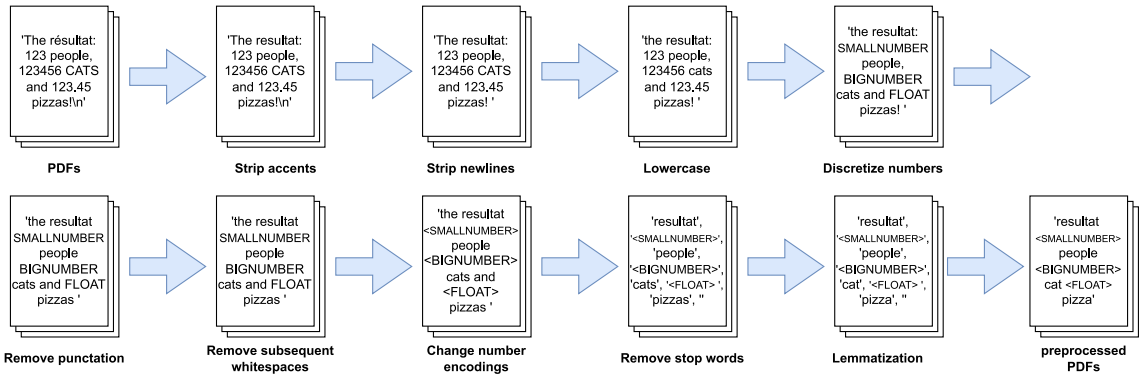


Figure 4.7: TF-IDF Preprocessing visualized using a example text.

4.6 Doc2Vec

The library `gensim` provides the Doc2Vec model used in this work. The input data to initialize the model has to be of type `tagged Documents`, which are documents with (numerical) tags. The parameter `dm` determines the training algorithm used. The value `dm=1` specifies the PVDM algorithm, while `dm=0` specifies the PV-DBOW algorithm [63]. The default algorithm, i.e. PVDM, is used in this work [27]. The parameters `vector_size` and `window` define the dimensionality of the embeddings and the size of the window, i.e. the maximum distance between the current and the predicted word, respectively. The default value for `vector_size` is 100, whereas the default window size is 8 [27, 26]. The `min_count` parameter defines a threshold below which words will be ignored. Its default value is 5. The `workers` parameter denotes the number of threads to be used for training. The default value is 1 [27]. The `epochs` parameter specifies the number of iterations over

the corpus. The default value is 10. By default, the hierarchical softmax algorithm, i.e. `hs=1`, is used for training [63]. Many Doc2Vec default values are adopted from Word2Vec since the `gensim` Doc2Vec implementation inherits from the Word2Vec implementation.

4.7 InferSent

The InferSent model is provided by PyTorch [48]. The parameters used to initialize the model are presented in Listing 4.8. The parameter `version` indicates whether the model was trained with GloVe or fastText for the value 1 or 2 respectively. Since the model is precomputed, it is not possible to change certain parameters, such as the word embedding dimension `word_emb_dim` or the dimension of the output vectors `enc_lstm_dim`.

```
{'bsize': 64, 'word_emb_dim': 300, 'enc_lstm_dim': 2048, 'pool_type': 'max',
'dpout_model': 0.0, 'version': 1}
```

Listing 4.8: Parameters of the InferSent model.

The steps necessary to create a working instance of the InferSent model are presented in Listing 4.9. After the InferSent model is initialized, the `state_dict` of the model is loaded. This dictionary consists of learnable parameters, i.e. weights and bias, of the model. The next step is to set the path to the word embeddings. Finally, the vocabulary of the model is built or more precisely, only those embeddings needed are kept while the rest is discarded.

```
infersent = InferSent(params_model)
infersent.load_state_dict(torch.load(model_path))
infersent.set_w2v_path(w2v_path)
infersent.build_vocab(docs, tokenize=True)
```

Listing 4.9: Initializing the InferSent model.

Initially, in this work, the InferSent model was based on GloVe word embeddings. GloVe is a global log-bilinear regression model for unsupervised learning of vector representations of words [45]. According to Pennington et al., the model captures global corpus statistics directly. The complexity of the model is bound by $O(V^2)$, V being the vocabulary size. However, Pennington et al. claim it is better than the worst case stated above and rather in $O(C)$, C being the corpus size. The GloVe model is trained on ratios of co-occurrence probabilities of words from the corpus, which correlate with the relationship between the words [45]. Pennington et al. introduce rules for a weighting function to ensure neither rare nor frequent co-occurrences are overweighted. It is possible to download embeddings computed by GloVe, instead of using the algorithm to generate them. The precomputed word embeddings are stored in a 5.65 GB text file. The file contains 840 B tokens and a

vocabulary of 2.2 M cased 300-dimensional vector representations of words [15]. According to Cer et al., GloVe introduces bias in terms of ageism, racism and sexism into the model.

In this work, a custom set of vector representations of words is used. The custom word embeddings are computed by a Word2Vec model trained on a selection of 195 documents from the Bahamas dataset. The only parameter which differs from the default settings is the `vector_size` which is set to 300. After the Word2Vec model is trained, the word embeddings are saved in a file. The file is post-processed to be compatible with the InferSent model. To be more precise, only lines that consist of at least two whitespace-separated char sequences are kept. Usually, word embeddings stored in a text file are structured in a way that the first char sequence is the word and the following numbers are the vector representation of the word.

In this work, an AE is used to reduce the dimensionality of the InferSent embedding. The implementation of the AE is outlined in section 4.4.

4.8 USE

The USE model is provided by TensorFlow [48]. In this work, the fourth version of the model is used. The implementation from Tfhub uses the DAN architecture [56]. It is about 1 GB and thus, loading the model takes a while. It is not necessary to preprocess the data for the model [56].

4.9 SBERT

The SBERT model is provided by PyTorch [48]. An instance of the model is obtained by initializing it as shown in Listing 4.10. The model consists of a BERT transformer, which has a `max_seq_length` of 128. It does not convert inputs to lowercase by default [47]. The output of the transformer is passed to a pooling layer, which is initialized with the `pooling_mode` parameter. The default is `mean_pooling`, which calculates the mean of the output vectors of the transformer. The other options are `cls_token_pooling`, which returns the output of the first token, `max_pooling`, which returns the maximum value of the output vectors, and `pooling_mode_mean_sqrt_len_tokens`. The word embedding dimension is 384 by default [47].

```
1 SentenceTransformer('paraphrase-MiniLM-L6-v2')
```

Listing 4.10: Initialization of the SBERT model.

4.10 Clustering using OPTICS

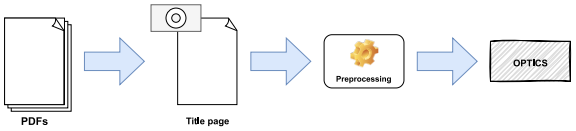


Figure 4.8: The first page of each document is converted to an image. The image is preprocessed, i.e. conversion to greyscale and resizing.

Similar to the approach from [5], OPTICS was used to cluster the images of the first page of documents in this work. The procedure is displayed in Figure 4.8. There were two different preprocessing approaches:

1. The images were first preprocessed to 32x32 normalized greyscale pixels (cf. [5]) as visualized in Figure 4.9 and afterwards compressed to 13-dimensional vectors using PCA.

2. The technique Eigendocs from subsection 3.5.2 was used to compress the images to 13-dimensional normalized greyscale images as displayed in Figure 5.1.

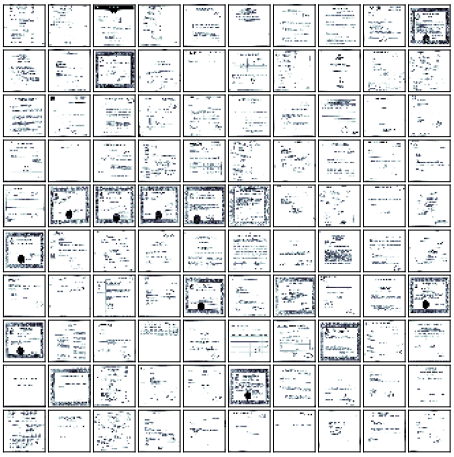
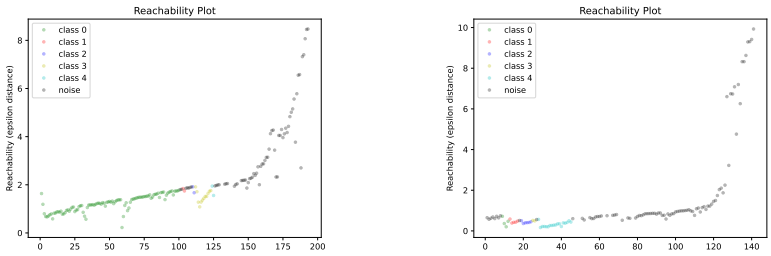


Figure 4.9: The first 100 documents of the dataset compressed to 32x32 greyscale pixels.

The reachability distance ordered by OPTICS is displayed in Figure 4.10. The resulting clusters are displayed in Figure 5.3.

The configurations used when initializing an OPTICS model greatly influence the clusters returned. The parameter `max_eps` is infinity by default but can be specified by the user to reduce complexity and runtime. According to literature, `max_eps` should be big enough to include almost all points in a cluster. The way the reachability plot is used to extract clusters is dependent on the `cluster_method`. One can choose either `dbscan` or `xi` as a clustering method. The parameters `min_samples` and `eps` influence the cluster sizes and number of clusters found for a given clustering approach. The value of `eps` defines the distance between two points to still be considered neighbours and can be chosen by



(a) The reachability plot of the documents preprocessed according to item 1.

(b) The reachability plot of the documents preprocessed according to item 2.

Figure 4.10: The plot was created using the OPTICS algorithm from the Python library scikit-learn. It shows the reachability distance of each document to its predecessor in the order list.

consulting the reachability plot. The code to initialize an exemplary OPTICS model is displayed in Listing 4.11.

```
optics_model = OPTICS(cluster_method='dbscan', min_samples=2, max_eps=10,
    eps=0.5)
```

Listing 4.11: Initialization of the OPTICS model. The number of minimum samples in a cluster corresponds to *minPts*.

4.11 top2vec

Angelov's top2vec model is provided in the Python library top2vec [4]. The non-changeable default values and settings of the model are listed below. The word and document embeddings are generated by the Doc2Vec version PV-DBOW [4]. It has a window size of 15, uses hierarchical softmax, a minimum count of 50, a vector size of 300 and a sub-sampling threshold of 10^5 [4]. UMAP's hyperparameters are set to 15 nearest neighbours, cosine similarity as the distance metric and 5 as the embedding dimension in Angelov's work.

In this work, a class is implemented, which uses the top2vec library. When initiating an instance of this class, the top2vec model is trained on the given document corpus as displayed in Listing 4.12. The class provides methods to query for the number of topics, most similar topics and documents to an input keyword. The most similar topics can be visualized using Word Clouds. The core functionalities are implemented by the top2vec library, but the class is used to modify the return values to be compatible with the UI.

```
Top2Vec(documents=self.documents, speed='fast-learn', workers=8)
```

Listing 4.12: Initialization of the top2vec model.

4.12 Word Cloud

The implementation of Word Clouds in this thesis is based on the Python library *wordcloud* [42]. This implementation removes English stop words from the text by default. The input text is split into tokens using a regex. By default, plurals are removed if their singular version is present and their frequency is added to their singular version. By default, numbers are not included as tokens.

In order to ensure that the words presented are interpretable, the input text is preprocessed. A lemmatizer is used to ensure stemmed words exist. The lemmatizer used is `WordNetLemmatizer` from the `nltk` package as displayed in Listing 4.13.

The Word Cloud is initialized as shown in Listing 4.14.

```
lemmatizer = WordNetLemmatizer()
tokens = [lemmatizer.lemmatize(token) for token in tokens]
```

Listing 4.13: Custom preprocessing of Word Cloud input.

```
wordcloud = WordCloud(width=800, height=500, random_state=21,
    contour_width=3, max_font_size=110, background_color='white',
    max_words=5000).generate(','.join(tokens))
```

Listing 4.14: Initialization of the Word Cloud.

4.13 User Interface

Since this work should be valuable to tax offices, a basic UI is provided. However, the focus of this work is on the methods and not on the UI. The UI is divided into two parts, the frontend and the backend.

4.13.1 Backend

The framework used for the backend is Flask. In this work, only the `GET` method is used. There are multiple endpoints, which are used to retrieve data from the server:

- Documents: Returns a list of documents, which best match the query. The query can be either of type `match_all`, which returns all documents in the database, or a fuzzy full-text query, or a kNN query on a certain field of the database. Moreover, the number and start index of the results returned can be specified.

- Document: Returns the document with the specified `id`.

- PDF: Returns the path to a PDF file. In order to access the path information a query for a document with the specified `id` is performed.

- WordCloud: Returns the bytes of a WordCloud image. Depending on additional parameters, the WordCloud is either generated from one document or the most similar documents to the query field, identified by kNN.

- Term Frequency: Returns the term frequency calculated for the specified document.

- TopicWordCloud: Returns a Word Cloud of the terms that describe the topics most similar to query term. The topics are generated by top2vec.

- Topics: Returns the topics generated by top2vec. The topics are described by the words closest to the topic vectors.

In order to test the endpoints during development, swagger documentation for every endpoint is provided.

4.13.2 Frontend

The framework used for the frontend is Angular. There are three main components, which are used to display the data:

- Home: The home component is used to display the results of a text query. It consists of a search bar, which is used to enter the query term, and a list of results. If no text query is entered the first documents of the database, i.e. the result of a `match_all` query, are displayed. The search component is shown in Figure 4.11.

- Detail: The detail component is used to display the details of a document. The document name and ID are located on the left side of the screen. Beneath the document name and ID, a button which opens the term frequency image on a new page upon pressing is located. Moreover, the WordCloud of the document is displayed. The WordCloud is generated from the text of the document. On the right side of the screen, there is a PDF viewer which displays the pages of the document. Beneath

the PDF viewer, the names and WordClouds of the most similar documents are displayed after a query for them is initiated by the user. The detail component is shown in Figure 4.12.

- Topic: The topic component is used to display the topics of the documents. The topics are generated by `top2vec`. The topic component is shown in Figure 4.13.

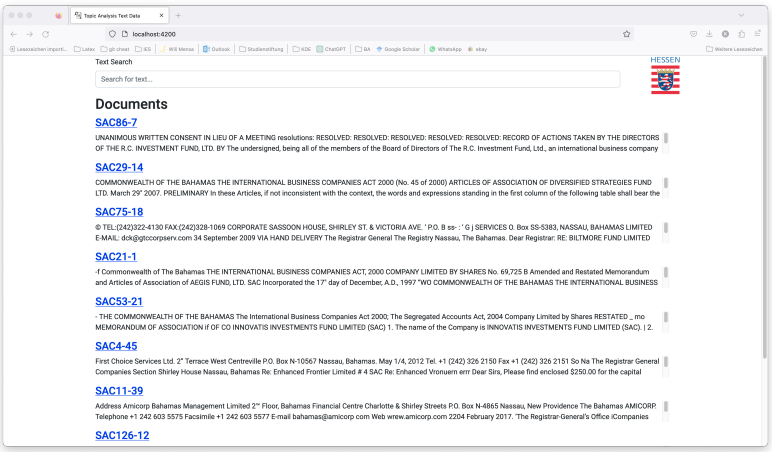


Figure 4.11: Home component of the frontend. The search bar is used to enter the text query. The results of the query are displayed below the search bar.

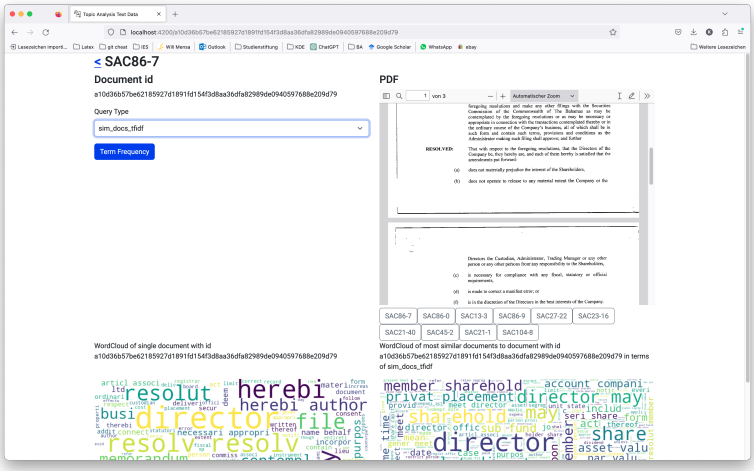


Figure 4.12: Detail component of the frontend. The chosen document is displayed, as well as its most similar documents in the database. WordClouds of the document and the most similar documents are displayed.

To change between the components, the routes have to be defined. The routes are defined in the `app-routing.module.ts` file, as shown in 4.15.

4.14 Trade-off between memory and query time

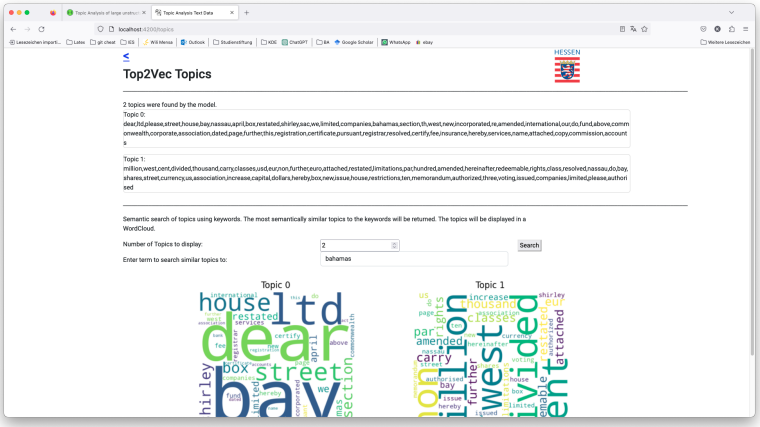


Figure 4.13: Topic component of the frontend. The topics identified by top2vec are listed. Below them, the user can query for the most similar topics to a term. The results are displayed as a Word Cloud.

```
const routes: Routes = [
    { path: 'topics', component: TopicsComponent},
    { path: ':id', component: DocumentDetailComponent},
    { path: '', component: HomeComponent},
  ];
```

Listing 4.15: Definition of routes in Angular in the `app-routing.module.ts`.

5 Evaluation

Parameters of models

5.1 Similarity measurements

According to Reimers and Gurevych, the similarity measurements discussed in section 3.2 obtained roughly the same results in their experiments [48].

5.2 Eigendocs

Assuming the layout holds information about the document type, the first page of each document is used to extract this information. In the course of working with low-quality versions of the documents to minimize the memory necessary to store them, some documents looked similar. Therefore, the idea arose to use clustering algorithms to group the documents according to their appearance.

In order to determine the optimal number of components used for Eigendocs the cumulative explained variance and the reconstruction error were plotted as displayed in Figure 3.10 from subsection 3.5.2. The first plot indicated that 90% of the variance is explained by 95 components. Usually, that would have been the number of dimensions of the subspace onto which the documents would have been projected. However, when working with cluster algorithms like OPTICS the number of dimensions should be reduced even further to achieve valid clusters. Therefore the second approach was used. The second plot showed the reconstruction error with respect to different numbers of components. The "elbow" points are visible at 10 and 13. Since visual inspection accounted for the fact that the decline of the reconstruction error after 13 declined more than after 10, the number of components chosen is 13.

The results of the Eigendocs algorithm are displayed in Figure 5.1.

The preprocessing of the documents using Eigendocs should have encoded information about the dimensionality of the images under the assumption the selection of documents is

Figure 5.1: The first 10 preprocessed documents of the dataset. The original images are displayed in the first row. The second row shows the reconstructed images using the compressed images from the fourth row. The third row shows the reconstruction error, i.e. the difference between the reconstructed and the original image. The last row presents the greyscale values of the compressed 13-dimensional image as a line.

representative. However, this assumption is not valid since there are bigger document images. Therefore, the idea of incorporating this information is not entirely implemented.

5.3 Evaluation of AE

In order to determine, which architecture for the hidden or so-called latent space of the AE is the best option, different architectures were tested and compared in terms of Root Mean Square Error (RSME) and cosine similarity. The RSME is calculated as given in Listing 5.1. The cosine similarity is calculated as given in Listing 5.2. The dataset used for the evaluation is a selection of 195 documents from the Bahamas dataset.

```
1 rsme = np.linalg.norm(inverse_embedding - embeddings)
2         / np.sqrt(embeddings.shape[0])
```

Listing 5.1: Computation of the RSME between the original and the reconstructed embedding.

```
1 cos_sim = statistics.mean([np.dot(inverse_emb, embedding)
2         /(np.linalg.norm(inverse_emb)*np.linalg.norm(embedding))
3         for inverse_emb, embedding in zip(inverse_embedding, embeddings)])
```

Listing 5.2: Computation of the cosine similarity between the original and the reconstructed embedding.

The scores of different architectures are shown in Figure 5.2. While most of the architectures produced similar results, one architecture stood out. Combining 2500, 3000 and 3500

dimensions in the hidden space produced the worst RSME results. The best results were achieved by adding 3500-dimensional layers in the hidden space. However, the results of the best architecture do not differ greatly from the others.

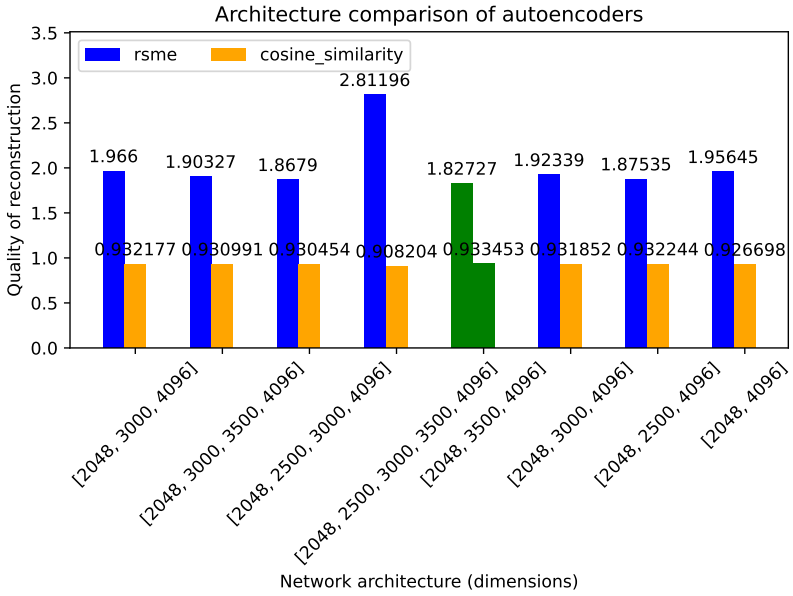
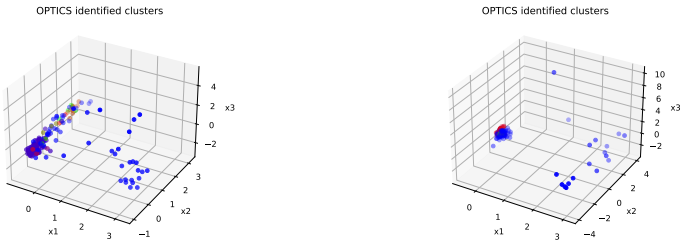


Figure 5.2: The effect of different AE architectures on the reconstruction error. The error is measured in terms of RSME (blue bars) and cosine similarity (yellow bars) between the original and the reconstructed image. The smallest RSME and the biggest cosine similarity belong to the architecture best suitable to this task and are coloured green.

5.4 Evaluation of OPTICS

The algorithm OPTICS was applied to data, which was preprocessed according to item 1 and item 2. The clusters from Figure 5.3 were extracted from the respective reachability plots in Figure 4.10. The three-dimensional plots visualize the first three dimensions of the data and thus, the weights of the first three principal components assigned by the Eigendocs algorithm. By visual inspection and comparison of both plots, it can be seen that the projection by the combination of resizing and PCA of item 1 scatters the objects further along the x_2 axis. Hence, the distance between the objects is larger and more clusters are identified. One could argue that the narrow distribution of the objects in the Eigendocs plot is due to the fact, that the input data encodes not only the visual appearance in terms of page layout but also the size of the document. Possibly, this could explain why the objects are less scattered along this dimension.

To analyse the results of the clustering, the content of the clusters was examined. Since the documents are not labelled, the content of the clusters was analysed by visual inspection. The content of the clusters is displayed in Figure 5.4 and Figure 5.5. The yellow images belong to the group identified as noise. The images preprocessed according to item 1 were



(a) Preprocessing according to item 1.

(b) Preprocessing according to item 2.

Figure 5.3: The clusters were extracted from the respective reachability plots in Figure 4.10 by OPTICS. The blue points are noise points, whereas any other colour denotes a cluster.

partitioned into multiple small and one big cluster. The Eigendocs images' clusters have similar sizes. The row of noise images is thus, way longer than the other rows in Figure 5.5. Most of the certificates are classified as noise for both approaches.

Figure 5.4: The yellow images belong to the group denoted noise. Most certificates are classified as noise. There is one big cluster and multiple small clusters. The images were preprocessed as discussed in item 1 to 32x32 greyscale pixels.

Figure 5.5: Most certificates are classified as noise. The rest of the clusters have similar sizes. The images were preprocessed as discussed in item 2 to 13-dimensional greyscale pixels.

The preprocessing approach used to create the OPTICS input for the Elasticsearch database index is Eigendocs since it also encodes information about the document size.

According to Deng et al., OPTICS was developed to improve DBSCAN flaws. Therefore, DBSCAN is chosen for the cluster method in Listing 4.11, since the literature consulted works with DBSCAN as a basis. In order to reduce calculation complexity the maximum ε is 10. The distance between two points to still be considered neighbours is defined after visual inspection of the reachability plot. Considering the intrinsic structure of the data it is set to 0.5 to return meaningful clusters.

5.5 Evaluation of database

According to [28], Structured Query Language (SQL) databases are a good choice for efficiently storing structured data. This is because their paradigm ACID, i.e. Atomicity, Consistency, Isolation, Durability, provides high reliability. Not only SQL (NoSQL) databases, on the other hand, are more flexible and can be used to store unstructured data [28]. They do not require a predefined schema and can therefore accept documents of arbitrary structure [25].

Usually, NoSQL databases do not offer services such as `JOIN` [25]. According to Gaspar and Stouffer, NoSQL databases make a tradeoff between storage and speed, as well as a tradeoff between consistency and availability. NoSQL databases are said to outperform out-of-the-box SQL databases [25].

The time necessary to perform the unique steps of filling the Elasticsearch database has been evaluated and improved over the course of this work. The current measurements are shown in Figure 5.6. Separating the initialization into multiple steps is beneficial since it is possible to update the embeddings without having to recreate the database. Moreover, modulizing the initialization and filling of the database facilitates debugging and comparing the models used to create the embeddings.

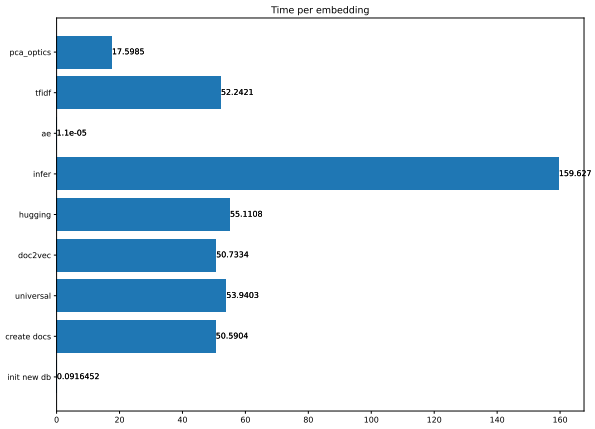
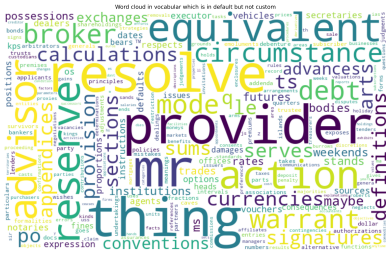


Figure 5.6: Time per module of creating the Bahamas database using a selection of 195 documents. The time was measured using the `timer` from `timeit.default_timer` on a Apple M2 Pro MNW83D/A with 16 GB RAM and 12 cores.

Since the similarity between vectors is usually calculated using some form of cosine similarity, rather than Euclidean distance in literature, cosine is preferred over Euclidean distance. Since the models may produce embeddings, which are not normalized, the cosine similarity is used instead of the dot product. Soft cosine similarity is not used, since it is not available in Elasticsearch. However, the usage of soft cosine would likely improve the results.



(a) The terms only present in the vocabulary produced by the default preprocessor.

(b) The terms only present in the vocabulary obtained from the custom preprocessor.

Figure 5.7: The Word Clouds visualize which words are not shared by both vocabularies.

A document store database can be used if the primary goal is to write fast rather than write save [25].

5.6 Evaluation of TF-IDF

The main obstacle to overcome is the high dimensionality of the TF-IDF embeddings. Hence, the goal of the parameter selection is to find a way to reduce the dimensionality of the vocabulary to the maximum vector dimensionality of Elasticsearch. However, the quality of the embeddings should not decline too much.

The choice of the preprocessor was investigated with regard to the goal of minimizing the vocabulary size. Both the default and a custom preprocessor were tested on a data corpus of 195 documents with regard to the vocabulary (size) and the result of preprocessing. A visualization obtained from the comparison is given in Figure 5.7. While the default preprocessor had a vocabulary size of 1641, the custom preprocessor had a size of 1521. The custom preprocessor was chosen because it had a smaller vocabulary size. The differences between both vocabularies were compared and visualized.

Initially, there should have been two different TF-IDF models. The first one should have been used to obtain documents which are similar to the query document. Therefore, terms which occur only once in the corpus should have been removed from the vocabulary. The second approach should have been used to obtain specific documents from the corpus. Hence, the vocabulary should consist of very document-specific terms and thus, `max_df` would have been relatively low, to omit terms that occur in many documents. However, the restrictions imposed by the database implementation made it impossible to explore many parameter ranges. Therefore, only one TF-IDF model was used in the end, whose parameters `min_df` and `max_df` were set to values which kept the vocabulary and thus, the dimensionality of the embeddings reasonably small.

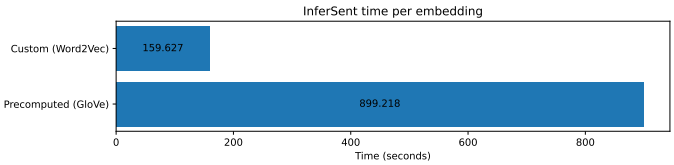


Figure 5.8: Time necessary to calculate and insert InferSent embeddings for different precomputed word embeddings on a Apple M2 Pro MNW83D/A with 16 GB RAM and 12 cores.

5.7 Evaluation of Doc2Vec

Since no labelled data is available, the evaluation of the Doc2Vec embeddings is limited. Therefore, the Doc2Vec embeddings are evaluated by comparing them to other embeddings. The Doc2Vec model is not tuned in terms of hyperparameter selection, but the default settings are used since there is no way to evaluate the resulting embeddings.

5.8 InferSent

The `max` pooling type is used for the InferSent model, since Conneau et al. found by conducting experiments using different pooling techniques that it was the best option.

Initially, in this work, the Global Vectors for Word Representation (GloVe) word embeddings were used for the InferSent model. However, since the file of precomputed GloVe word embeddings has a size of 5.65 GB and thus, slows down the model, ultimately another word embedding was used. The time necessary to compute and insert 195 documents for specific embeddings is displayed in Figure 5.8. The custom word embedding used in this work is a Word2Vec model trained on a selection of 195 documents from the Bahamas dataset.

Pennington et al. state that GloVe outperforms Word2Vec on the same corpus, vocabulary and window size in terms of quality and speed [45]. Hence, the quality of the results obtained in this work may have suffered from using a custom Word2Vec instead of GloVe. However, since the computation time of the project is a crucial factor, the custom Word2Vec was used.

5.9 Evaluation of USE

Since there are no parameters to customize the evaluation of the USE embeddings is limited. Therefore, the USE embeddings are evaluated by comparing them to other embeddings.

5.10 analysis/ comparison of models

Similar to Pennington et al.'s work, in this work, for many models used, any unspecified parameters are set to their default values, assuming that they are close to optimal acknowledging that this simplification should be revised in a more thorough analysis.

difference query responses for different models? any images which produce unusual results?

5.11 Evaluation of the performance

5.11.1 Fahnder clustern

5.11.2 Fahnder bewerten Resultate (image matrix)

5.12 Evaluation of the usability

5.12.1 Metrics

6 Results

Evaluate the results from the previous chapter.

6.1 Fulfilment of objective

6.2 Research results

7 Conclusion

8 Outlook

8.1 Future Work

Bibliography

[1] Quick start user guide. URL `https://slurm.schedmd.com/quickstart.html`. [Accessed 16.09.2023].

[2] K.P. Agrawal, Sanjay Garg, Shashikant Sharma, and Pinkal Patel. Development and validation of optics based spatio-temporal clustering technique. *Information Sciences*, 369:388–401, 2016.

[3] Rubayyi Alghamdi and Khalid Alfalqi. A survey of topic modeling in text mining, 2015.

[4] Dimo Angelov. Top2vec: Distributed representations of topics. 2020.

[5] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. Optics: Ordering points to identify the clustering structure. *SIGMOD Rec.*, 28:49–60, 1999.

[6] Farzana Anowar, Samira Sadaoui, and Bassant Selim. Conceptual and empirical comparison of dimensionality reduction algorithms (pca, kpca, lda, mds, svd, lle, isomap, le, ica, t-sne). *Computer Science Review*, 40:100378, 2021.

[7] Fankar Armash Aslam, Hawa Nabeel Mohammed, Jummal Musab Mohd. Munir, and Murade Aaraf Gulamgaus. Efficient way of web development using python and flask. *International Journal of Advanced Research in Computer Science*, 6:54–57, 2015.

[8] Steven Bird, Ewan Klein, and Edward Loper. *Natural Language Processing with Python*. O'Reilly Media, Sebastopol, CA, USA, 1st edition, 2009.

[9] Daniel Cer, Yinfei Yang, Sheng yi Kong, Nan Hua, Nicole Limtiaco, Rhomni St. John, Noah Constant, Mario Guajardo-Cespedes, Steve Yuan, Chris Tar, Yun-Hsuan Sung, Brian Strope, and Ray Kurzweil. Universal sentence encoder, 2018.

[10] Delphine Charlet and Géraldine Damnati. Simbow at semeval-2017 task 3: Soft-cosine semantic similarity between questions for community question answering, 2017.

[11] Alexis Conneau, Douwe Kiela, Holger Schwenk, Loic Barrault, and Antoine Bordes. Supervised learning of universal sentence representations from natural language inference data, 2018.

[12] Matt Copperwaite and Charles Leifer. *Learning Flask Framework*. Packt Publishing, 2015.

[13] Laura Dayton, Dante Rousseve, Neil Sehgal, and Sindura Sriram. Csci 1430 final project report: Methods of facial recognition, 2020.

[14] Z. Deng, Y. Hu, M. Zhu, and et al. A scalable and fast optics for clustering trajectory big data. 18:549–562, 2014.

[15] download-glove. Glove: Global vectors for word representation. URL `https://nlp.stanford.edu/projects/glove/`. [Accessed 04.10.2023].

[16] Elasticsearch-fuzziness. Fuzziness. URL `https://www.elastic.co/guide/en/elasticsearch/reference/current/common-options.html#fuzziness`. [Accessed 15.09.2023].

[17] Elasticsearch-get. Get api. URL `https://www.elastic.co/guide/en/elasticsearch/reference/current/docs-get.html`. [Accessed 15.09.2023].

[18] Elasticsearch-kNN. k-nearest neighbor search. URL `https://www.elastic.co/guide/en/elasticsearch/reference/current/knn-search.html`. [Accessed 15.09.2023].

[19] Elasticsearch-kNN-embedding. How to deploy a text embedding model and use it for semantic search. URL `https://www.elastic.co/guide/en/machine-learning/8.10/ml-nlp-text-emb-vector-search-example.html`. [Accessed 15.09.2023].

[20] Elasticsearch-kNN-similarity. Dense vector field type. URL `https://www.elastic.co/guide/en/elasticsearch/reference/current/dense-vector.html#dense-vector-similarity`. [Accessed 15.09.2023].

[21] Elasticsearch-match. Match query. URL `https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-match-query.html`. [Accessed 15.09.2023].

[22] Elasticsearch-synonyms. Synonyms. URL `https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-match-query.html#query-dsl-match-query-synonyms`. [Accessed 15.09.2023].

[23] Elasticsearch-text-analyser. Text analysis overview. URL https://www.elastic.co/guide/en/elasticsearch/reference/current/analysis-overview.html. [Accessed 15.09.2023].

[24] Fabio. Deep averaging network.ipynb. URL https://github.com/f0bs/Machine_Learning/blob/master/Deep%20Averaging%20Network.ipynb4. [Accessed 04.10.2023].

[25] Daniel Gaspar and Jack Stouffer. *Mastering Flask Web Development: Build Enterprise-Grade, Scalable Python Web Applications,*, volume 2. Packt Publishing, 2018.

[26] gensim-doc2vec-init. gensim.models.doc2vec. URL https://tedboy.github.io/nlps/generated/generated/gensim.models.Doc2Vec.__init__.html. [Accessed 01.10.2023].

[27] gensim-word2vec-init. Word2vec embeddings. URL https://radimrehurek.com/gensim/models/word2vec.html#gensim.models.word2vec.Word2Vec. [Accessed 01.10.2023].

[28] Miguel Grinberg. *Flask Web Development: Developing Web Applications with Python.* O'Reilly Media, Inc., 2018.

[29] Klara M. Gutekunst. Identifying fiscal fraud with anomaly detection techniques. Technical report, University of Kassel, 2023. URL https://github.com/KlaraGtknst/identifying-fiscal-fraud.

[30] Hari Krishna Kanagala and V.V. Jaya Rama Krishnaiah. A comparative study of k-means, dbscan and optics. In *2016 International Conference on Computer Communication and Informatics (ICCCI)*, pages 1–6, 2016.

[31] Pooja Kherwa and Poonam Bansal. Topic modeling: A comprehensive review. *EAI Endorsed Transactions on Scalable Information Systems*, 7(24), 7 2019. doi: 10.4108/eai.13-7-2018.159623.

[32] Gunjan Khosla, Navin Rajpal, and Jasvinder Singh. Evaluation of euclidean and manhanttan metrics in content based image retrieval system. In *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 12–18, 2015.

[33] Matt J. Kusner, Yu Sun, Nicholas I. Kolkin, and Kilian Q. Weinberger. From word embeddings to document distances, 2014.

[34] Tzu-Hsuan Lin and Jehn-Ruey Jiang. Credit card fraud detection with autoencoder and probabilistic random forest. *Mathematics*, 9(21), 2021. doi: 10.3390/math9212683. URL `https://www.mdpi.com/2227-7390/9/21/2683`.

[35] Yu. A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs, 2018.

[36] Tomas Mikolov and Quoc Le. Distributed representations of sentences and documents, 2014.

[37] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

[38] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality, 2013.

[39] Sumit Misra, Soumyadeep Thakur, Manosij Ghosh, and Sanjoy Kumar Saha. An autoencoder based model for detecting fraudulent credit card transaction. *Procedia Computer Science*, 167:254–262, 2020. doi: https://doi.org/10.1016/j.procs.2020.03.219. International Conference on Computational Intelligence and Data Science.

[40] Mauritius Much, Frederik Obermaier, Bastian Obermayer, and Vanessa Wormer. So funktioniert das system bahamas. URL `https://www.sueddeutsche.de/wirtschaft/bahamas-leaks-so-funktioniert-das-system-bahamas-1.3172913`. [Accessed 08.08.2023].

[41] Mohammad Robihul Mufid, Arif Basofi, M. Udin Harun Al Rasyid, Indhi Farhandika Rochimansyah, and Abdul rokhim. Design an mvc model using python for flask framework development. In *2019 International Electronics Symposium (IES)*, pages 214–219, 2019.

[42] Andreas Müller. word cloud. URL `https://github.com/amueller/word_cloud/tree/main`. [Accessed 05.10.2023].

[43] Li-Qiang Niu and Xin-Yu Dai. Topic2vec: Learning distributed representations of topics. 2015.

[44] Mostofa Ali Patwary, Diana Palsetia, Ankit Agrawal, Wei-keng Liao, Fredrik Manne, and Alok Choudhary. Scalable parallel optics data clustering using graph algorithmic techniques. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, New York, NY, USA, 2013. Association for Computing Machinery.

[45] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation, 2014.

[46] Robert-George Radu, Iulia-Maria Rădulescu, Ciprian-Octavian Truică, Elena-Simona Apostol, and Mariana Mocanu. Clustering documents using the document to vector model for dimensionality reduction, 2020.

[47] Nils Reimers and Iryna Gurevych. sentence-transformers/paraphrase-minilm-l6-v2. URL `https://huggingface.co/sentence-transformers/paraphrase-MiniLM-L6-v2#sentence-transformersparaphrase-minilm-l6-v2`. [Accessed 04.10.2023].

[48] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks, 2019.

[49] Shyam Seshadri. *Angular Up and Running: Learning Angular, Step by Step*. O'Reilly Media, Inc., 2018.

[50] Grigori Sidorov, Alexander Gelbukh, Helena Gómez-Adorno, and David Pinto. Soft similarity and soft cosine measure: Similarity of features in vector space model, 2014.

[51] Gerd Stumme, Robert Jäschke, and Christoph Scholz. Internet-suchmaschinen, 2011.

[52] Dodi Sudiana, Mia Rizkinia, and Fahri Alamsyah. Performance evaluation of machine learning classifiers for face recognition. In *2021 17th International Conference on Quality in Research (QIR): International Symposium on Electrical and Computer Engineering*, pages 71–75, 2021.

[53] Yichuan Tang and Xuan Choo. Intrinsic divergence for facial recognition, 2008.

[54] tfidf-scikit-learn. Tf–idf term weighting. URL `https://scikit-learn.org/stable/modules/feature_extraction.html#text-feature-extraction`. [Accessed 29.09.2023].

[55] M. A. Turk and A. P. Pentland. Face recognition using eigenfaces. *Proc. IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 586–591, 1991.

[56] UniversalSentEnc-dev. universal-sentence-encoder. URL `https://tfhub.dev/google/universal-sentence-encoder/4`. [Accessed 04.10.2023].

[57] A. Voit, A. Stankus, S. Magomedov, and I. Ivanova. Big data processing for full-text search and visualization with elasticsearch, 2017.

[58] Chang Wang and Sridhar Mahadevan. Multiscale manifold learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 27:912–918, 2013.

[59] Andy B. Yoo, Morris A. Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Job Scheduling Strategies for Parallel Processing*, pages 44–60, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[60] V. Zamfir, M. Carabas, C. Carabas, and N. Tapus. Systems monitoring and big data analysis using the elasticsearch system, 2019.

[61] Jun Zhang, Yong Yan, and M. Lades. Face recognition: eigenface, elastic matching, and neural nets. *Proceedings of the IEEE*, 85(9):1423–1435, 1997.

[62] Wen Zhang, Taketoshi Yoshida, and Xijin Tang. Tfidf, lsi and multi-word in information retrieval and text categorization, 2008.

[63] Radim Řehůřek. Doc2vec paragraph embeddings, 2022. URL `https://radimrehurek.com/gensim/models/doc2vec.html`. [Accessed 01.10.2023].

List of Figures

List of Tables

Listing-Verzeichnis

A Anhang

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur mit den nach der Prüfungsordnung der Universität Kassel zulässigen Hilfsmitteln angefertigt habe. Die verwendete Literatur ist im Literaturverzeichnis angegeben. Wörtlich oder sinngemäß übernommene Inhalte habe ich als solche kenntlich gemacht.

Kassel, October 17, 2023

Klara Maximiliane Gutekunst