

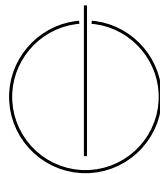
DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

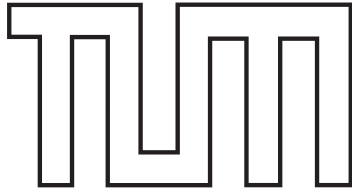
Master's Thesis in Informatics

**Label Propagation for Tax Law Thesaurus  
Extension**

Markus Müller







DEPARTMENT OF INFORMATICS

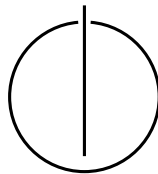
TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

# Label Propagation for Tax Law Thesaurus Extension

## Label Propagation zur Erweiterung von Steuerrechtsthesauri

Author: Markus Müller  
Supervisor: Prof. Dr. Florian Matthes  
Advisors: Prof. Dr. Stephan Günemann  
Jörg Landthaler, M. Sc.  
Elena Scepankova, Mag. jur.  
Submission Date: November 7th, 2018



I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, November 7th, 2018

Markus Müller

## Acknowledgments

I especially would like to thank my advisors Jörg Landthaler and Prof. Stephan Günemann for their support. The topics I address in this thesis - machine learning, data science and legal tech - are so diverse and full of different options. I always could be sure of their guidance in choosing the right direction. I am very glad that, one and a half years ago, I decided to join Jörg Landthaler's seminar "Natural Language Processing for Law". I liked the seminar and Jörg's supervision very much and soon recognized that I wanted to dive deeper into the topic. Thanks to my supervisor, Prof. Florian Matthes, for making this work at his chair possible!

Thanks to Aleksandar Bojchevski from the Professorship of Data Mining and Analytics for providing me with insight about the label propagation algorithms and confirming my doubts about sklearn's implementation. Thanks to Prof. Günemann for providing me with the infrastructure to run my computation-heavy experiments. Thanks to Alexander Schell for confirming my proof on using euclidean distance instead of cosine distance for better performance. Thanks to Christina Dosch for her continuous support and understanding.



# Abstract

With the rise of digitalization, information retrieval has to cope with increasing amounts of digitized content. Legal content providers invest a lot of money for building domain-specific ontologies such as thesauri to retrieve a significantly increased number of relevant documents. Since 2002, many label propagation methods have been developed e.g. to identify groups of similar nodes in graphs. Label propagation is a family of graph-based semi-supervised machine learning algorithms. In this thesis, we will test the suitability of label propagation methods to extend a thesaurus from the tax law domain. The graph on which label propagation operates is a similarity graph constructed from word embeddings. We cover the process from end to end and conduct several parameter-studies to understand the impact of certain hyper-parameters on the overall performance. The results are then evaluated in manual studies and compared with a baseline approach.

**Keywords** Thesaurus Extension, Legal Tech, Information Retrieval, Label Propagation, Word Embeddings, Data Science, Machine Learning





# Contents

<b>Acknowledgments</b>	<b>iii</b>
<b>Abstract</b>	<b>v</b>
<b>1. Introduction</b>	<b>1</b>
1.1. Problem Statement . . . . .	3
1.2. Problem Solution . . . . .	4
1.3. Research Process . . . . .	7
<b>2. Prerequisites and Related Work</b>	<b>9</b>
2.1. Data Set . . . . .	9
2.1.1. Text Corpus . . . . .	9
2.1.2. Thesaurus . . . . .	9
2.1.3. Usage . . . . .	11
2.2. Distributional Semantics . . . . .	11
2.2.1. Word Embedding Technologies . . . . .	11
2.2.2. Thesaurus Creation and Extension . . . . .	13
2.3. Graph Construction . . . . .	14
2.4. Label Propagation . . . . .	16
2.4.1. Semi-supervised Learning in General . . . . .	16
2.4.2. Label Propagation Intuition . . . . .	17
2.4.3. Input Data . . . . .	18
2.4.4. LabelPropagation . . . . .	19
2.4.5. LabelSpreading . . . . .	21
2.4.6. Applications . . . . .	23
<b>3. Implementation</b>	<b>25</b>
3.1. Pipeline Architecture . . . . .	25
3.2. Pipeline Filters . . . . .	27
3.2.1. Corpus Pre-Processing . . . . .	27
3.2.2. Embeddings Generation . . . . .	29
3.2.3. Graph Construction . . . . .	31
3.2.4. Thesaurus Pre-Processing . . . . .	32
3.2.5. Thesaurus Sampling . . . . .	34
3.2.6. Graph Labeling . . . . .	35
3.2.7. Label Propagation . . . . .	35

3.2.8. Evaluation Filter . . . . .	36
<b>4. Quantitative Evaluation</b>	<b>37</b>
4.1. Structure . . . . .	37
4.2. Parameter Studies . . . . .	38
4.2.1. Embeddings Generation . . . . .	39
4.2.2. Graph Construction . . . . .	40
4.2.3. Propagation Phase . . . . .	41
4.2.4. Pre-Processing . . . . .	43
4.3. Optimized Configurations . . . . .	44
<b>5. Qualitative Evaluation</b>	<b>47</b>
5.1. Pre-Study . . . . .	48
5.2. Main Study . . . . .	50
<b>6. Assessment and Further Refinements</b>	<b>53</b>
6.1. Baseline: k-nearest-neighbors of Synset Vector . . . . .	53
6.1.1. Quantitative and Qualitative Evaluation . . . . .	54
6.1.2. Prediction Similarity Comparison . . . . .	57
6.1.3. Application of Propagation to Baseline . . . . .	59
6.2. Challenges around Thesaurus Extension . . . . .	60
<b>7. Conclusion and Future Work</b>	<b>63</b>
<b>A. Proof: Euclidean instead of Cosine Distance for Word Embeddings</b>	<b>67</b>
<b>B. Difference between two LabelPropagation Algorithms by Zhu et al.</b>	<b>69</b>
<b>List of Figures</b>	<b>71</b>
<b>List of Tables</b>	<b>73</b>
<b>Bibliography</b>	<b>75</b>

# 1. Introduction

In today's society, the amount of available digital information is continuously growing. Finding the information one needs out of this massive amount of data is called *information retrieval*. More formally, Schütze et al. (2008) define information retrieval as

...finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers).

In this thesis, we focus on text as source of information. Nowadays, almost everyone with a digital device engages in information retrieval multiple times per day. Typical information retrieval tasks include the use of web search engines and searching one's email. Often, the more ambiguous word "search" is used as a synonym for "(information) retrieval". Schütze et al. (2008) distinguish information retrieval systems into three different scales: *Web search*, where the system has to provide search over billions of documents. *Personal information retrieval*, such as Apple's macOS Spotlight or email search, where the system provides search over one's personal documents. And in between, the space of *enterprise, institutional or domain-specific search*, where retrieval might be provided over a company's internal documents or e.g. specific research database.

Just as society as a whole evolves into an "information society", information-intensive industries and related tasks within the existing industries have become more important. Under the term "knowledge-based economy", OCDE (1996) recognized knowledge and information as the driver of productivity and economic growth. The ability of coping with lots of information becomes more and more crucial for economic success. People are expected to dig through masses of text to find the relevant information. Jobs where information retrieval is central include "analyst" functions like research analysts within consultancies, and mergers & acquisitions analysts in the finance industry.

**In the Legal Domain** The focus of this thesis is the *legal domain* in which information retrieval systems are central for many tasks as well. We can identify two stakeholders: First, we have *lawyers and attorneys* whose work typically is knowledge-based. For a given case, they need to gather as much information as possible, e.g. on the context itself<sup>1</sup>, on the relevant laws, on previous judgments and on assessments by other

---

<sup>1</sup>For cases around corporations, this includes the need to collect and analyze relevant information from within the company.

lawyers. Comments around judgments and legislation are often summarized in legal journals. The second group of stakeholders, *legal content (and media) providers*, support their customers, who are mostly lawyers and attorneys, in their work by providing relevant content. They maintain large databases with legal documents and provide their customers access to these databases. Examples in Germany include DATEV<sup>2</sup>, Haufe<sup>3</sup>, Wolters Kluwer<sup>4</sup>, C.H. Beck<sup>5</sup> and ottoschmidt<sup>6</sup>. Access is provided via search. When a user enters a search query, he expects to be provided with *all relevant documents*. This is a major challenge in information retrieval and receives much attention also in the legal domain, cf. Tamsin Maxwell (2009), Conrad and Lu (2013), and Grabmair et al. (2015). Efforts in improving legal search systems are counted to the general area of *Legal Tech*, which refers to the use of technology and software to provide legal services.

**Full-text search as a means for information retrieval** In the beginning of computerized information retrieval during the late 1940s, document retrieval was always based on meta-data like author, title, and keywords, but not on the actual full-text of documents itself (Schütze et al. 2008). Later, full-text search was employed. Traditional full-text search finds *exact* matches to a given search string, corresponding to the search query (Landthaler et al. 2016). But discovering all relevant documents still remained challenging. As an example from the legal world, the term “Abwrackprämie” (engl. scrapping incentive) refers to the bonus Germans received in 2009 when they scrapped their old car in order to buy a new one. While “Abwrackprämie” was primarily used by the media, the term used by the government was “Umweltprämie”. A full-text search for “Abwrackprämie” will therefore not return all documents related to the scrapping incentive concept. We focus on this issue of *synonymy* (Schütze et al. 2008) - a concept may be referred to using different words. The search query should be refined to match documents that include either of the two terms.

**Thesauri for Query Expansion** To address the synonymy issue, information retrieval systems try to help the user in expanding the query. For example, they could include synonyms in the query automatically, or suggest these synonyms or related words to the user for manual selection. These approaches are called *query expansion* (Schütze et al. 2008). The most common form of query expansion uses some form of *thesaurus*. Within the context of search query expansion, we view a thesaurus as a collection of *synonym sets* (“synset”). A synset consists of words that express a common distinct *concept*. In general, a thesaurus can contain sets of different word relations, not just synonyms. Examples are antonyms, abbreviations, broader or narrower terms.

---

<sup>2</sup><https://www.datev.de>, visited on Nov. 3, 2018

<sup>3</sup><https://www.haufe.de>, visited on Nov. 3, 2018

<sup>4</sup><https://www.wolterskluwer.de>, visited on Nov. 3, 2018

<sup>5</sup><https://www.chbeck.de>, visited on Nov. 3, 2018

<sup>6</sup><https://www.otto-schmidt.de>, visited on Nov. 3, 2018

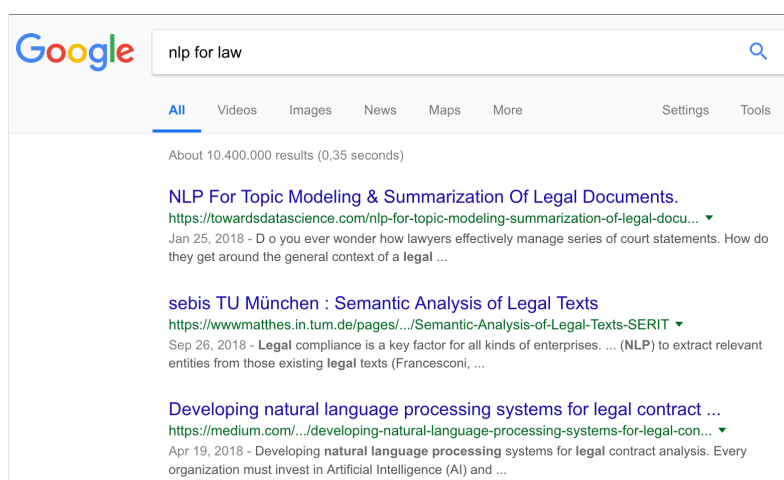


Figure 1.1.: Demonstration of an automatic search query expansion by Google. For the query “nlp for law”, documents with the terms “legal” and “natural language processing” were retrieved as well.

Figure 1.1 shows such an automatic search query expansion for a search with Google: Although the terms “legal” and “natural language processing” were not part of the query (“nlp for law”), the query was expanded to these terms as well, as can be seen in the retrieved results and the bold highlighting of these new terms.

Query expansion is a widely used method to increase the recall<sup>7</sup> of search systems. Thesauri are a common way to implement query expansion. Dirschl (2016) from the legal content provider Wolters Kluwer call legal thesauri the “backbone of many application features in JURION [their content platform]”.

## 1.1. Problem Statement

We have seen that thesauri are a useful means to improve a search system’s recall. The downside is that creation and maintenance of such thesauri is a laborious, expensive and error-prone task. Although there exist some large thesauri as WordNet (Miller et al. 1990), these general-purpose thesauri are usually not suited for domain-specific tasks. Even one common thesaurus for the legal domain is seen as too broad - Dirschl (2016) state they have multiple smaller, domain-specific thesauri in place, e.g. for areas like intellectual property law or construction law.

<sup>7</sup>Recall is the fraction of relevant instances that have been retrieved over the total amount of relevant instances. As an example: If there are 10 relevant instances in total, a system that retrieved 9 of these relevant instances would have a recall of  $\frac{9}{10} = 0.9$ . As search provider, we want our recall to be high.

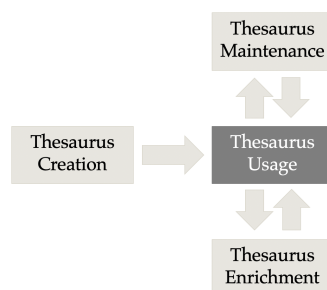


Figure 1.2.: The main activities around thesauri according to Dirschl (2016).

**What is needed** At least since the 1950s, there have been attempts by the natural language processing community to make the process around thesaurus creation more automated (Harris 1954; Sparck Jones 1964). Figure 1.2 shows the activities around thesauri derived from Dirschl (2016). First, a thesaurus gets created in a mostly manual process. It can get enriched, which means that it gets mapped and linked to other standard thesauri in order to exploit existing public knowledge. Thesaurus maintenance deals with the continuous update process, i.e. creating new synsets, adding new words to existing ones or regroup them. All these phases could be supported with sophisticated technology in an automatic or semi-automatic fashion. It could automatically create synsets or suggest additions to already manually started synsets. As part of thesaurus enrichment, it could suggest links between synsets of different thesauri. It could become a part of the thesaurus maintenance task, e.g. to notify the maintainer when new words could be added to the synsets or existing synsets should be adjusted.

## 1.2. Problem Solution

In this thesis, we pursue an approach for finding similar words to *extend existing synsets*. We combine two technologies: On the one hand, we use *Word Embeddings*, multi-dimensional vector representations for words, where similar words are placed close to each other. On the other hand, our second device is *Label Propagation (LP)*, a family of semi-supervised machine learning algorithms that propagates information from labeled words to unlabeled words. Both technologies are extensively used in practice, but have not been used together for thesaurus creation yet. We are particularly motivated from the work at Google (Ravi and Diao 2015)<sup>8</sup>, where word embeddings and label propagation are used together on large-scale data to learn emotion associations.

We focus on the use case of supporting thesaurus creation. Concretely, the creator of a thesaurus first manually has to build a minimal thesaurus with all synsets they want

---

<sup>8</sup>Also see this Google AI Blog post: <http://ai.googleblog.com/2016/10/graph-powered-machine-learning-at-google.html>, visited on Nov. 3, 2018

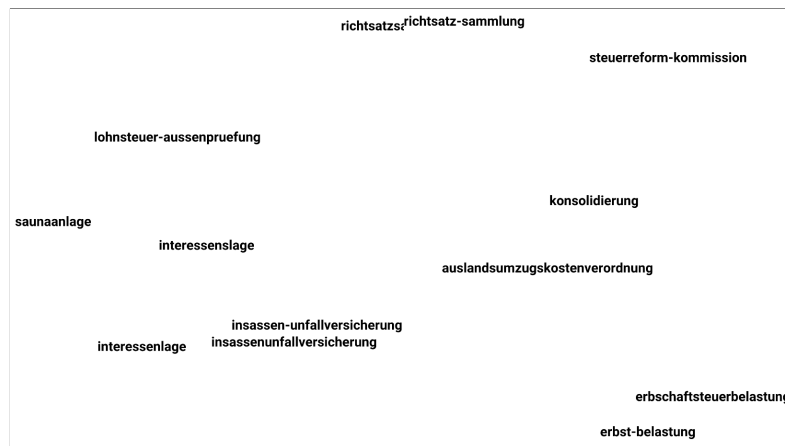


Figure 1.3.: Sample word embeddings for a German tax law corpus visualized. Similar words appear close to each other. The embeddings are projected down to two dimensions (originally: 400 dimensions).

to support. Then, our approach suggests additions to these existing synsets. These candidates need to be reviewed and can then be added to the thesaurus, which makes the approach a semi-automatic one. In the future, this approach could be extended to other parts of the process shown in Figure 1.2, e.g. by taking the respective word suggestions into account when determining the links between synsets of different thesauri, or regularly calculating word suggestions when the underlying text corpus has changed.

Combining word embeddings with label propagation for thesaurus extension has not been tried before. With this project, we evaluate whether this combination gives promising results for further research.

**Word Embeddings** Our approach is motivated by the improvements in word embedding technologies in the recent years. Word embeddings are vector representations of words that are related to their semantic meaning. The more similar two words, the closer their word vectors.<sup>9</sup> The computation is based on the distributional hypothesis by Harris (1954): Words which occur in the same contexts are likely to have a similar meaning. More details are given in Section 2.2.1. Although word embeddings have been subject to research since at least the 1980s, especially the work by Mikolov et al. (2013a) on “word2vec” made it possible to calculate high-quality embeddings even on large corpora. Word embeddings are useful finding new synonymous words. Words that are close to existing words in a synset are very promising candidates. Figure 1.3 shows some sample word embeddings from existing synsets visualized. We can easily observe that similar words appear close to each other. Landthaler et al. (2017) evaluated

<sup>9</sup>Measured in cosine distance.

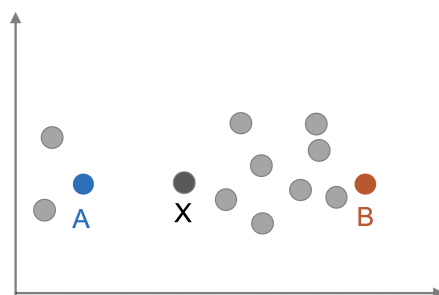


Figure 1.4.: Word embeddings A and B are labeled with different synsets, the other embeddings are unlabeled. With a nearest neighbors approach, X would receive A's synset. But from the overall structure, one would assume that X should be added to B's synset.

the feasibility of the word embedding approach for thesaurus extension and reported promising results.

But also word embeddings have to be handled with care. Figure 1.4 shows an issue with simply using the existing synset's nearest neighbors as synset candidates. The figure shows some schematic word embeddings. Words A and B are labeled with different synsets, the other words are not labeled. The goal now is to determine which unlabeled word should become part of which synset. A simple way would be selecting the nearest neighbors of each labeled word. But in case of the situation shown in Figure 1.4, this would lead to an unintuitive result. X is closer to A than to B and therefore would receive the same label as A. But X has many close neighbors that themselves would be labeled with B's label. Although X is not in the direct neighborhood of B, assigning it the same label as B might be more intuitive. By taking *the overall structure* into account, the suggestions would have been better. Label propagation methods can solve this problem.

**Label Propagation** Label propagation is a family of semi-supervised graph-based machine learning algorithms (Bengio et al. 2006). It has promising characteristics for our use case of extending existing synsets.

- **Semi-supervision** Semi-supervised algorithms are used to predict labels for unlabeled data. They use a small amount of labeled data and a large amount of unlabeled data as training data. For us, this small amount of labeled data corresponds to the words in the thesaurus which already have been assigned to a synset. And we have a large amount of unlabeled data: The words in the text corpus that are not part of the thesaurus. Supervised algorithms like the nearest neighbors method shown in Figure 1.4 just take labeled data into account. A semi-supervised approach like label propagation would likely consider the small distance between the unlabeled embeddings and assign all of them to the same label.



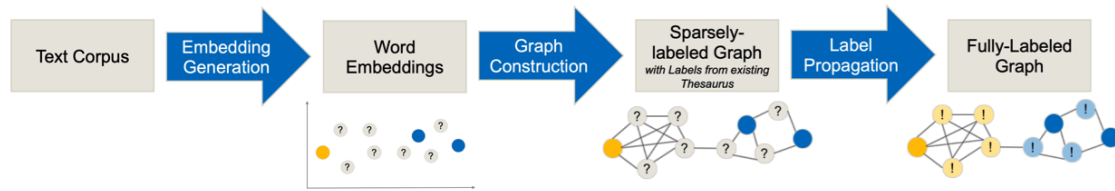


Figure 1.5.: Intuition of Combining Word Embeddings with Label Propagation for assigning unlabeled words to existing synsets

- **Scalability** Label propagation algorithms can be made scale well to large datasets and lots of labels. In a blog post by Google<sup>10</sup>, they note that they have been able to scale their approaches to billions of nodes, trillions of edges involving billions of different label types. Our data set involves around 200,000 nodes and 2000 label types. In other areas, the number of nodes and labels can be significantly higher.

Label propagation algorithms work on graphs. The graph's structure needs to be correlated with the classification goal. In our case, words correspond to graph nodes. Similar words need to be connected via a small number of edges or edges with low weight. Word embeddings are multi-dimensional vectors that need to be converted to a graph so label propagation can be applied. How to do this graph construction is also subject to our research.

**Combination Intuition** Figure 1.5 depicts the intuition in combining word embeddings with label propagation for Thesaurus Extension. First, word embeddings for a text corpus are generated. The existing thesaurus synsets act as labels. The embeddings with their labels are converted to a sparsely-labeled graph. With label propagation, labels for the previously-unlabeled nodes are determined. This labeling corresponds to predicting to which synset a word should be added, i.e. how a synset should be extended.

### 1.3. Research Process

**Research Questions** We formulated five research questions that we tried to answer during the course of this thesis:

1. How can we get semantic & context information into a graph for LP?
2. How can we model the thesaurus extension problem on the LP technology?
3. Which LP algorithms work best?
4. Is LP a suitable technology for thesaurus extension in the legal domain?
5. How much automation for thesaurus creation is achievable with LP?

<sup>10</sup><http://ai.googleblog.com/2016/10/graph-powered-machine-learning-at-google.html>, visited on Nov. 3, 2018

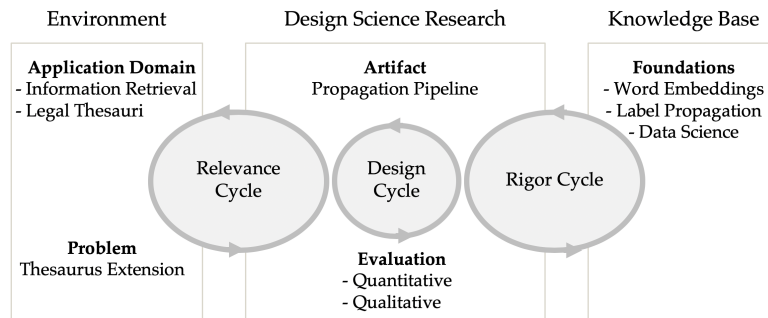


Figure 1.6.: Our design science research process visualized

To answer our research questions, we employed a Design Science process (Hevner and Chatterjee 2010) which is depicted in Figure 1.6. We have identified the thesaurus extension problem for legal information retrieval. From data science research, we selected the word embedding and the label propagation technologies. The design artifact, a pipeline architecture, covers the phases from an existing thesaurus to producing candidates for its extension. We need an existing thesaurus that we extend, as well as a text corpus where we can generate candidates from. For that, we were provided with a data set on tax law by the legal content provider DATEV. It consists of a text corpus (130,000 legal documents) and a handcrafted thesaurus (16,000 concept classes). The thesaurus was specifically created to support full-text search on the text corpus.

We defined performance metrics and set up a test environment by using parts of the thesaurus as test data. There, we evaluated and compared it with baseline approaches. We used this information to iteratively adjust our pipeline architecture (Design Cycle). Also, we continuously extended the architecture with alternative word embedding and label propagation approaches to evaluate them as well (Rigor Cycle). After optimizing our architecture based on the quantitative evaluation, we conducted a manual qualitative evaluation. There, we rated the perceived quality of the algorithmic suggestions.

**Structure** In Chapter 2, we describe the data set in detail and give an overview on word embedding technologies, label propagation approaches and previous thesaurus extension research. There, we answer our first research question, how to get semantic & context information into a graph for label propagation. We line out the pipeline architecture in Chapter 3. This answers the second research question on how to model the thesaurus extension problem. In the Quantitative Evaluation, In Chapter 4, we derive an optimized set of hyper-parameter values. This answers our third research question. We present the results of our manual Qualitative Evaluation in Chapter 5. In Chapter 6, we compare our label propagation method with a rather simple baseline approach. The answers to the two remaining research questions are implied by conclusion in Chapter 7. We then discuss potential drawbacks of our approach and directions for future work.

## 2. Prerequisites and Related Work

### 2.1. Data Set

We use a large corpus of German tax law, accompanied by a thesaurus that is specifically maintained for that corpus. The data is provided by an industry partner of our chair: DATEV eG, a technical legal services provider that is focused on tax matters. It was already used in previous research by Landthaler et al. (2017).

#### 2.1.1. Text Corpus

The text corpus contains 132,581 legal documents of different document types. An overview of the distribution of different document types is shown in Figure 2.1. The documents are stored as JSON files. Unprocessed, including all meta-data (e.g. author, document type, topic, title), these files add up to 3.6 GB. We discard meta-data and focus on the documents' full text. All full texts together make up 170,065,082 tokens, out of which 2,215,785 tokens are unique. A token consists of characters separated by whitespace. More statistics can be found in Table 2.1.

	Minimum	Median	Maximum	Mean	Std. Deviation
<i>Tokens</i>	14	452	304,131	1282.74	4771.65

Table 2.1.: Corpus document size statistics (rounded to two decimals).

#### 2.1.2. Thesaurus

The thesaurus consists of 16,019 concept classes. It holds six different types of relations (e.g. hyponyms and abbreviations), however we focus on the 12,288 synonym concepts. These synsets lead to 36,076 keys, with 35,502 unique keys. We use the term “*key*” instead of “*word*” to denote the entries of a synset, as entries sometimes consist of multiple words, e.g. “Zusatz Tarifvertrag”. Synset sizes range from 2 to 32, with 2 being the most common size. Table 2.2 shows more statistics on the synset sizes, Figure 2.2 shows a histogram of the synset sizes in the untouched thesaurus.

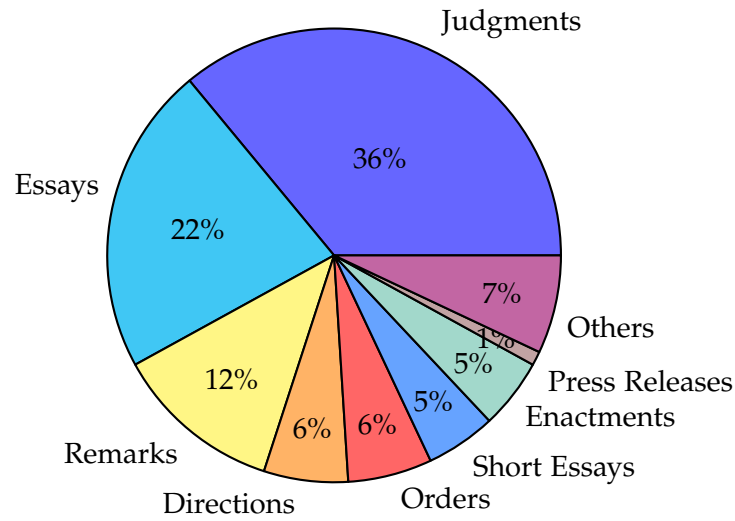


Figure 2.1.: The distribution of the different document types that can be found in the provided text corpus, taken from Landthaler et al. (2017).

	Minimum	Median	Maximum	Mean	Std. Deviation
<i>Concept Size</i>	2	2	32	2.94	1.58

Table 2.2.: Thesaurus synset size statistics (rounded to two decimals).

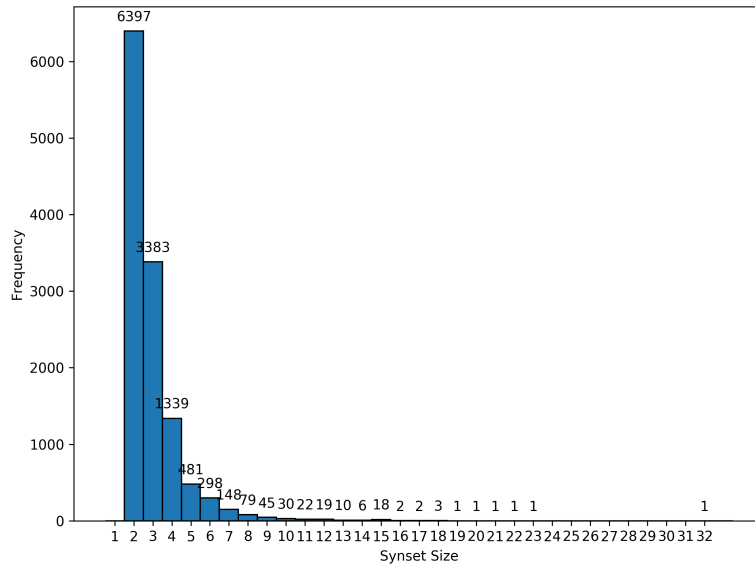


Figure 2.2.: Histogram visualizing synset sizes in the untouched thesaurus.

### 2.1.3. Usage

The text corpus is used to calculate semantic similarities between its words. The thesaurus' existing synsets are used as training data. From the training data and via the word similarity relations, we predict additions to the synsets. For the quantitative evaluation, we split each synset in the thesaurus into a 50% training and 50% test part. For the qualitative evaluation, we use the full thesaurus as training data and rate the suggestions' quality manually. Both, text corpus and thesaurus, need to be prepared (pre-processed) to be used in the thesaurus extension. Pre-processing will be discussed in detail in Chapter 3, *Implementation*.

## 2.2. Distributional Semantics

We use word embeddings to represent the semantics of words in vector space. Word embeddings apply a *Distributional Semantics* approach. The contexts in which a word occurs, become key to derive a word meaning's representation (Clark 2015). This is also called the *distribution* of the word's contexts. Distributional semantics are based on the *Distributional Hypothesis* by Harris (1954): Words that occur in the same contexts tend to be perceived as similar by humans. An Example: From the two sample contexts "I watched the game" and "We watched the game" we could infer that "I" and "We" are semantically similar. Of course, many more contexts are needed to infer actual word similarities. The definition of "context" varies according to the particular technique being used. A popular approach is to select an  $n$  words context window and slide this window across the whole corpus. Each of the  $n$ -term word sets is then regarded as a context.

We first introduce the word embedding technologies we use in our work. Then, we show research applying distributional semantics to automate thesaurus creation and extension.

### 2.2.1. Word Embedding Technologies

By applying the distributional semantics approach, words from a text corpus can be *embedded* into a multi-dimensional vector space. Words are mapped to vectors of real numbers according to their semantic characteristics. These vectors are called *word embeddings* and were initially presented by Hinton et al. (1986). Three popular word embedding technologies that we use in our work are: *word2vec*, developed at Google by Mikolov et al. (2013a), *GloVe*, developed at Stanford University by Pennington et al. (2014), and *fastText*, developed at Facebook by Bojanowski et al. (2017).

Word embeddings represent the semantics of words. We expect words with similar meaning to have similar vectors. In the three models, the distributional similarity

between two word vectors is measured by the *Cosine Similarity*, the cosine of their enclosing angle (interval  $[0, 1]$ ). By subtracting the cosine similarity from 1, we get the *Cosine Distance*.<sup>1</sup> The more similar two words, the lower the cosine distance. We exploit this fact for finding synonyms.

Baroni et al. (2014) use the term “Distributed Semantic Models” (DSMs) for what we call word embedding technologies. They distinguish between two main DSM families for learning word vectors: 1) *context-counting* models and 2) *context-predicting* models.<sup>2</sup>

Counting models are the more traditional way - they construct a (large) co-occurrence count matrix (words correspond to rows, selected contexts correspond to columns). A value in the matrix is the count of how often a word appears in a given context. From there, dimensionality reduction techniques are applied to generate a lower-dimensional (less columns) matrix where a row corresponds to a vector representation for the respective word. A classical count-based DSM is *Latent Semantic Analysis* (LSA) by Deerwester et al. (1990), GloVe is a newer one (Pennington et al. 2014).

Predictive DSMs either predict the context a word tends to appear, or predict a word from its neighbors. Word vectors correspond to *weights* of a neural network. The word vectors are initialized randomly. With these initial vectors as weights, the neural network prediction performance will be poor. The values are adjusted iteratively to minimize the prediction error. With each iteration, the word vectors are refined and the model prediction performance increases. They were developed by the neural-network community (Bengio et al. 2003) and popularized by the works of Mikolov et al. (2013a) on word2vec.

**word2vec** word2vec, a predictive DSM, became very successful because it provides two comparatively simple architectures (shallow, two-layer neural networks), called CBOW and Skip-gram, that have much lower computational complexity compared to the previous popular (deep) neural network models. Despite their simplicity and shallowness, these architectures are able to generate high quality word vectors. Due to the lower computational complexity, it is possible to compute very accurate high dimensional word vectors from larger data sets. The CBOW architecture tries to predict a word from its context, while the Skip-gram architecture predicts surrounding words given the current word, as is depicted in Figure 2.3. An additional advantage compared to previous models is that arithmetic calculations tend to correspond to word analogies. For example, *King - Man + Woman* corresponds to a vector very close to *Queen* (Mikolov et al. 2013b).

---

<sup>1</sup>Cosine Distance is not a proper distance metric in the mathematical sense as it does not fulfill the triangle inequality, but within our use case, this is not an issue.

<sup>2</sup>Baroni et al. (2014) limit the term “word embeddings” to the context-predicting type of model. Similarly to other research (Landthaler et al. 2016; Lebrecht and Collobert 2014), we use the term “embedding” more generally. We express that words get *embedded* into a vector space.

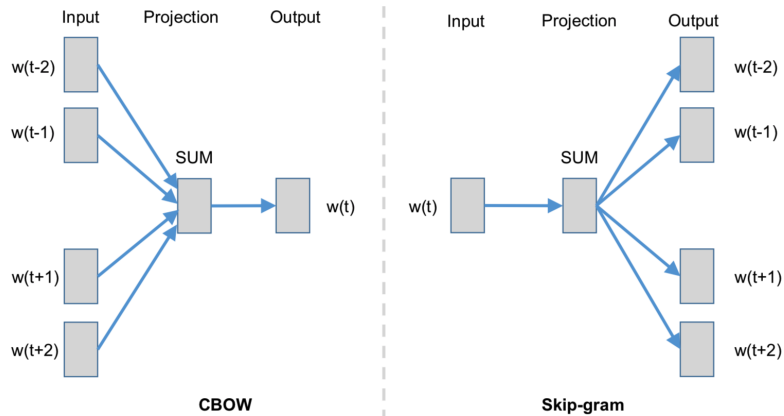


Figure 2.3.: word2vec Continuous Bag-of-words (CBOW) and Skip-gram training model architectures, from Mikolov et al. (2013a).

**fastText** This DSM builds on the word2vec model with the important difference that, instead of treating words as smallest entity, it operates on a sub-word level. Each word is now seen as a composite of  $n$ -grams,  $n$  consecutive characters (with a  $min\_n$  and  $max\_n$  set). For  $min\_n=3$  and  $max\_n=6$ , the word *where* would be represented by: “<wh”, “whe”, “her”, “ere”, “re>”, “<whe”, “wher”, “here”, “ere>”, “<wher”, “where”, “here>”, “<where”, “where>”. In addition, the word itself, “<where>”, is added an entity. “<” and “>” are added as word boundary characters to distinguish prefixes and suffixes from other character sequences. Each of these  $n$ -grams receives a vector representation. This leads to better performance in particular for rare words because  $n$ -grams are shared between words. Additionally, word vectors for completely unknown words can be constructed after the training phase, by combining the  $n$ -gram vectors needed to build up the word.

**GloVe** GloVe is a count-based “response” to the grown popularity of word2vec. One of its goals was to generate vectors where arithmetic operations express meaning as well. Pennington et al. (2014) claim to consistently outperform word2vec, with even lower training time.

## 2.2.2. Thesaurus Creation and Extension

**Thesaurus Creation** Even though thesaurus creation is not the main focus of this thesis, we briefly cover this topic. It has already been an interesting research field for several decades and provides the groundwork for later works.

There has been considerable research in automating the thesaurus creation process via distributional similarity approaches. Early approaches include Sparck Jones (1964),

who investigated how count-based relations can help in grouping words similar to a thesaurus. Salton (1989) mentions automatic thesaurus construction approaches in order to improve *document vector* generation. Grefenstette (1994) gives a broad overview on the history and need for automatic thesaurus discovery.

*Sketch Engine* by Rychlý and Kilgarriff (2007) is focused on increasing the efficiency of generating thesauri from *large data sets*. They automatically identify and remove word pairs that have nothing in common. To identify “useless” word pairs, they take their grammatical relation into account. This pruning phase enabled them to process a dataset with 2 billion words in less than 2 hours, compared to 300 days without the removal.

Kiela et al. (2015) argue that word embeddings capture *both* similarity and relatedness, two often incompatible objectives. In specializing embeddings for similarity, they show improvements for synonym selection. Word embeddings also have been trained to capture antonym relations (Ono et al. 2015; Nguyen et al. 2016), a word relation that is also often contained in thesauri. With *AutoExtend*, Rothe and Schütze (2015) proposed an extension for word embeddings: Synset embeddings, where a vector corresponds to a whole synset. Synset vectors then live in the same vector space and can be compared to words and other synsets via the usual cosine similarity measure. The system can be extended to other word relation concept like antonyms and just needs regular word embeddings as input. The concept of “synset embeddings” is an inspiration for our baseline approach defined in Section 6.1.

**Thesaurus Extension** Several methods and ideas for automatic thesaurus extension have been tried out over the years. Examples include Uramoto (1996), Takenobu et al. (1997), and Meusel et al. (2010). Landthaler et al. (2017) targeted the extension of existing synsets in a legal thesaurus via word embeddings. They proposed an intersection method, where they first train multiple word2vec embedding sets, each time with different parameters, and then, when determining the synonyms for a certain target word, intersect the result lists of the different embeddings sets. To select the appropriate embedding sets to intersect, they introduce the *RP-Score*. It measures how close members of existing synsets are positioned to each other. However, in our work we want to try out a new concept which has not been part of the scientific discussion yet, namely combining word embedding techniques with label propagation algorithms for semi-supervised machine learning. For that, word embeddings need to be represented as a graph.

### 2.3. Graph Construction

In order to apply to label propagation to our problem of thesaurus extension, we need to construct a graph out of the text corpus’ word embeddings. A word embedding then corresponds to a graph node. The graph should reflect the neighborhood relationships between the embeddings. It needs to represent the pairwise similarities/distances. Zhu



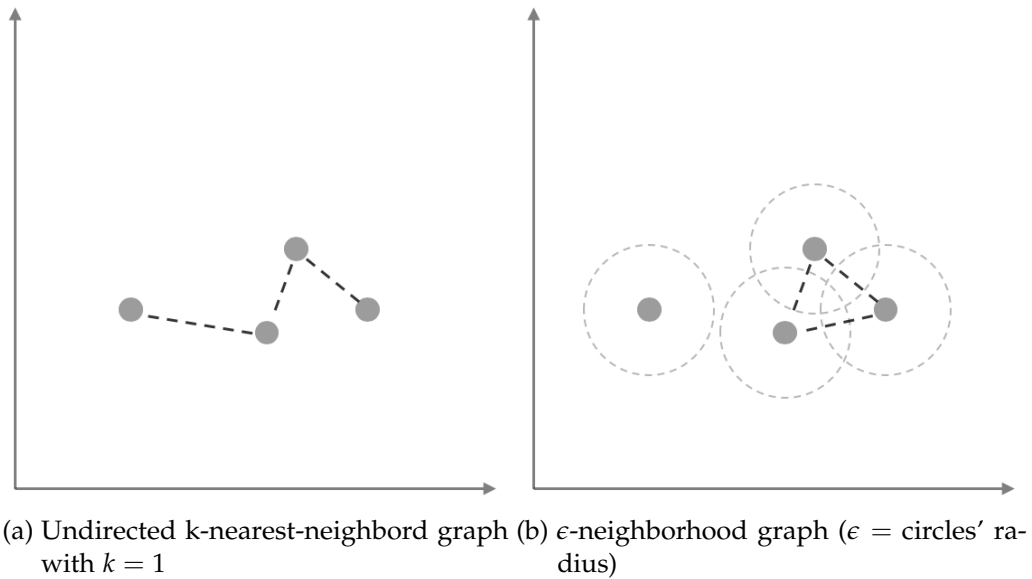


Figure 2.4.: Sample  $k$ -nearest-neighbor and  $\epsilon$ -neighborhood graph construction in a two-dimensional setting.

et al. (2005) and von Luxburg (2007) mention several approaches for the construction of such a *similarity graph*. Here, we focus on two well-known concepts:  *$k$ -nearest-neighbors graph*, and  *$\epsilon$ -neighborhood graph*. Figure 2.4 shows sample graphs for some given points in a two-dimensional vector space.

**$k$ -nearest-neighbors graph** An item  $x_i$  gets connected with another item  $x_j$  via an edge if  $x_j$  is among the  $k$ -nearest neighbors of  $x_i$ . This definition usually leads to a directed graph.<sup>3</sup> To make our graph undirected, we ignore directions. If one of two nodes is in each other's  $k$ -neighborhood, we add edges in both directions. This is usually called *the  $k$ -nearest-neighbor graph* (von Luxburg 2007).<sup>4</sup>

**$\epsilon$ -neighborhood graph** Here, we connect all points whose pairwise distances are smaller than  $\epsilon$ . This graph is symmetric by definition. Note that for this graph it is possible for nodes to not have any edges, while in a  $k$ -nearest neighbors graph, every node will have at least  $k$  edges. On the other side, many nodes might end up with lots of edges, if  $\epsilon$  is large.

<sup>3</sup>Example: When thinking of three 1-dimensional points  $A = 1, B = 2, C = 10$ ,  $C$ 's nearest neighbor is  $B$ , but  $B$ 's is  $A$ . The resulting graph would be directed.

<sup>4</sup>Another option is to ignore edges that are not mutual - this is called the *mutual  $k$ -nearest neighbor graph*. We did not investigate this option further.

**Weight Matrix** A graph can be stored as a weight matrix - a  $n \times n$  matrix  $W$  where  $W_{ij}$  corresponds to the weight of the edge between node  $i$  and  $j$ .<sup>5</sup> If the weight is 0, the two nodes are not connected. If all weights are 1, the graph is called *unweighted*. If weights differ from each other, the graph is called *weighted*. We can weight edges e.g. by the similarity of the nodes. In case of word embeddings, this would be the cosine similarity value.

## 2.4. Label Propagation

The “Label Propagation” approach was introduced by Zhu and Ghahramani (2002). It is a family of algorithms that deals with the semi-supervised learning problem of learning from labeled and unlabeled data (Bengio et al. 2006).

### 2.4.1. Semi-supervised Learning in General

In semi-supervised learning, the goal is to predict the labels of unlabeled points. Semi-supervised learning algorithms like label propagation are good at solving problems where two consistency assumptions apply (Zhou et al. 2004):

1. **Local consistency assumption.** *Nearby* points are likely to have the same label.
2. **Global consistency assumption.** Points on the *same structure* are likely to have the same label.

Supervised learning algorithms like k-nearest-neighbors<sup>6</sup> in general consider only the first assumption of local consistency. An illustrative example from Zhou et al. (2004) is shown in Figure 2.5. According to the consistency assumption, the data should be classified as two half-moons. A semi-supervised algorithm can solve this problem correctly.

In supervised learning, an algorithm tries to infer a function *from labeled training data* that generalizes well to unseen data. In contrast, to fulfill the global consistency assumption, semi-supervised learning algorithms use a *mixture* of labeled and unlabeled data as training data, where most of the data is unlabeled. Semi-supervised algorithms make use of the structure of the overall data - *including the unlabeled data*. Semi-supervised algorithms either predict labels for new unknown data (*inductive learning*) or label the training data itself (*transductive learning*). Label propagation performs transductive learning. This fits our “thesaurus extension” scenario well. The text corpus words are mapped to word embeddings - the structure of the word embeddings is used to infer synset labels to the text corpus words themselves.

---

<sup>5</sup>A weight matrix is also called *adjacency matrix* or *affinity matrix*.

<sup>6</sup>There, a point gets labeled with the label that occurs most often within his  $k$  nearest labeled neighbors.

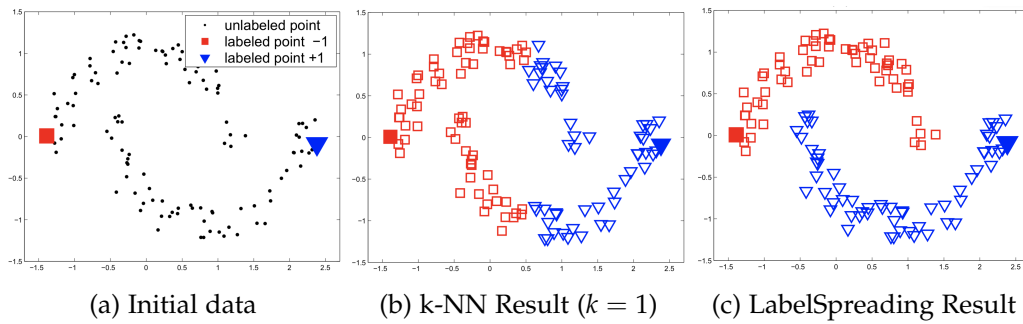


Figure 2.5.: Classification comparison from Zhou et al. (2004): k-nearest-neighbors does not take global consistency into account, LabelSpreading (a label propagation approach) does.



Figure 2.6.: Intuition of how in label propagation algorithms, label information flows out of the labeled nodes into unlabeled ones.

## 2.4.2. Label Propagation Intuition

Semi-supervised algorithms differ in how they realize the global assumption condition. For label propagation algorithms, the intuition is that “labeled data act like sources that push out labels through unlabeled data” (Zhu and Ghahramani 2002). Figure 2.5 visualizes this intuition. Label information is iteratively flowing out of the labeled nodes (yellow and blue) into the unlabeled ones. We note two structures: The yellow label gains control over the left structure, the blue one over the right structure.

Label propagation algorithms do not operate directly on multi-dimensional vectors, but on graphs. The data first needs to be converted into a graph structure, where the graph structure correlates with the goal of classification. In case for thesaurus synset extension, a similarity graph (see Section 2.3) needs to be constructed. Note that there is not “the one” label propagation algorithm, but instead the term describes a family of algorithms that all propagate labels along the graph structure.

We describe the two basic label propagation algorithms that have been subject to most research: *LabelPropagation* by Zhu et al. (2005), and *LabelSpreading* by Zhou et al. (2004). We will then give an overview of where label propagation has been applied in general. To distinguish between “label propagation”, the family of algorithms that propagate labels, and the specific algorithm by Zhu et al. (2005), we write the latter as “LabelPropagation”. For consistency, we refer to the algorithm by Zhou et al. (2004) as “LabelSpreading”.

### 2.4.3. Input Data

The input data for LabelPropagation and LabelSpreading is prepared in the same way. We describe the process along the lines of Bengio et al. (2006). The data is represented by a graph  $G = (V, E)$ , where nodes  $V = \{1, \dots, n\}$  represent the training data (labeled and unlabeled) and edges  $E$  represent similarities between them. The similarities are given by a *weight matrix*  $W$ .  $W_{ij}$  is non-zero iff  $x_i$  and  $x_j$  are neighbors, i.e. the edge  $(i, j) \in E$  (weighted by  $W_{ij}$ ). The graph is usually assumed to be undirected<sup>7</sup> and positive (Bengio et al. 2006; Buchnik and Cohen 2017). For LabelPropagation, it is not clearly defined whether self-referencing edges should be allowed. Bengio et al. (2006) claim preventing them ( $W_{ii} = 0$ ) often works better. For LabelSpreading, Zhou et al. (2004) explicitly state that there should be no self-referencing edges.

All nodes' labels get saved in a *label distribution matrix* (or *class probability matrix*)  $Y$  with shape  $(\#nodes, \#classes)$  - each row corresponds to a node, each column to a specific class.  $Y^{(k)}$  corresponds to the label distribution matrix after the  $k$ -th iteration ( $Y^{(0)}$  for the initial,  $Y_i^{(\infty)}$  for the converged distribution). Nodes  $\{1, 2, \dots, l\}$ , where the label is known, get labeled with a one-hot encoding vector  $y_i$ , i.e. 0 everywhere except 1 where the index corresponds to the class of  $x_i$ . Nodes  $\{l + 1, \dots, n\}$  get labeled with a zero-value vector. In this explanation, we order labeled nodes before unlabeled nodes - this is just for better overview; in general, a specific node order is not required. Note that we assume a multi-class classification scenario, i.e.  $\#classes > 2$ . If the classification is binary,  $Y$  can have shape  $(\#nodes, 1)$  and the nodes can simply be labeled with 1 or  $-1$  for known labels and 0 for unknown label. Per node, the value in each column can be interpreted as *confidence* for the respective class.

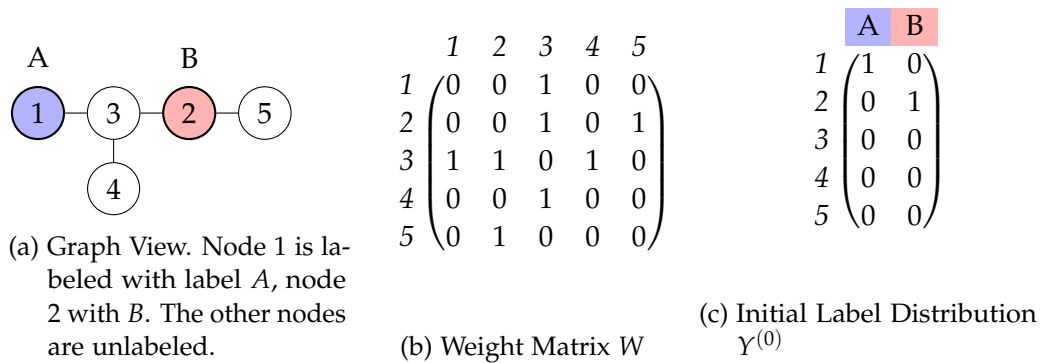


Figure 2.7.: An exemplary graph  $G$  with five nodes (two labeled, three unlabeled)

**Example** For an exemplary graph  $G$  with weight matrix  $W$  and initial label distribution  $Y^{(0)}$ , see Figure 2.7. Node 1 is labeled with class A, node 2 with class B, the other nodes

<sup>7</sup>This means the weight matrix is symmetric.

are unlabeled. Looking forward, one can intuitively see that node 5 should receive the same label as 2, whereas the label for 3 and 4 is undetermined, as both labeled nodes influence them. When presenting the LabelPropagation and LabelSpreading algorithms, we will come back to this example and show how each of them determines a final label distribution.

#### 2.4.4. LabelPropagation

---

**Algorithm 1:** LabelPropagation (Zhu and Ghahramani 2002)

---

Compute weight matrix  $W$

Initialize  $Y^{(0)} \leftarrow (y_i, \dots, y_l, 0, 0, \dots, 0)$

Compute the (weighted) diagonal degree matrix  $D$  by  $D_{ii} \leftarrow \sum_j W_{ij}$

Compute transition matrix  $T \leftarrow D^{-1}W$

Iterate

1.  $Y^{(t+1)} \leftarrow TY^{(t)}$

2.  $Y_l^{(t+1)} \leftarrow Y_l^{(0)}$  // Reset learned labels of labeled nodes

until convergence to  $Y^{(\infty)}$

Label point  $x_i$  with the class that resembles the index of the highest value in  $Y_i^{(\infty)}$

---

In LabelPropagation (algorithm 1), after calculating the input matrices, the transition probability matrix  $T$  gets computed by multiplying inverse (weighted) diagonal degree matrix  $D^{-1}$  and weight matrix  $W$ .<sup>8</sup> This corresponds to letting the labels “flowing in” via adjacent edges, with the amount being determined by the respective edge weight in relation to the other adjacent ones. The new class probabilities  $Y^{(t+1)}$  are then calculated by applying the transition matrix  $T$  to  $Y^{(t)}$ .

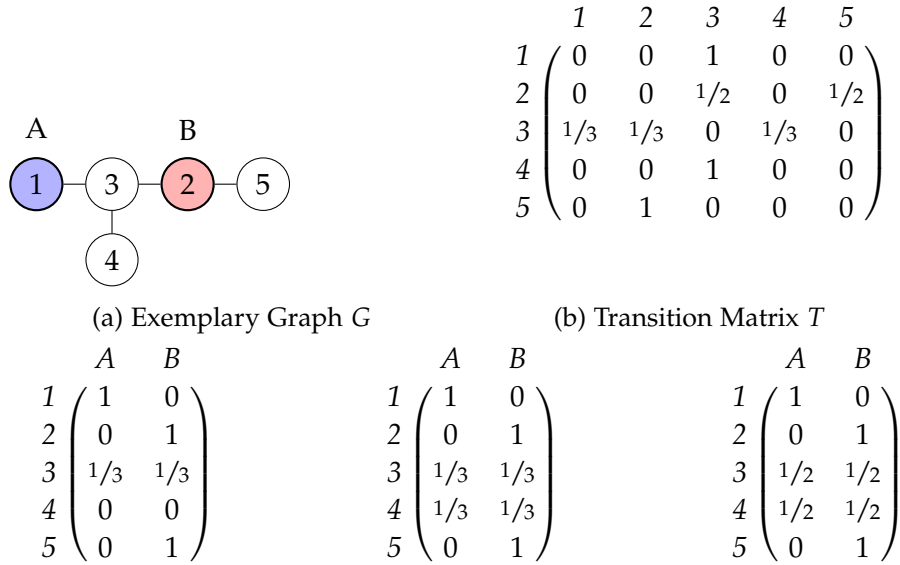
Then, the learned labels of the already initially labeled nodes get reset. This is also called *clamping*. If we would not reset these labels, we would not get a steady stream of label information. The initial label information would just “fade away”, potentially with allocating a different label to the initially labeled nodes.

The application of the transition matrix  $T$  on the current label distribution matrix  $Y^{(t)}$  and clamping is repeated until convergence. Then, each node gets labeled with the most likely (highest-valued) class from the node’s row,  $Y_i^{(\infty)}$ .

**Example** To come back to our exemplary graph  $G$ , Figure 2.8 shows the transition matrix  $T$ , the first two iterations of  $Y$  the final label distribution matrix. Node 5 gets labeled with class  $B$ , while for nodes 3 and 4, both possible classes get assigned the same probability. This corresponds to our intuition from before.

---

<sup>8</sup>The inverse of a diagonal matrix can be simply calculated by inverting each matrix value.



(c)  $Y^{(1)}$ , label distributions after first iteration (d)  $Y^{(2)}$ , label distributions after second iteration (e)  $Y^{(\infty)}$ , converged label distributions

Figure 2.8.: *LabelPropagation* transition matrix for graph  $G$  and iteratively calculated label distributions for its nodes

**Different LabelPropagation Algorithms** Between the original paper (Zhu and Ghahramani 2002) and the papers referencing it (Bengio et al. 2006; Buchnik and Cohen 2017), there is a slight difference in the algorithm described. The probabilistic transition matrix is calculated differently. Bodó and Csató (2015) investigate this issue in detail. Without further note, widely-cited papers like Bengio et al. (2006) actually describe the algorithm from Zhu et al. (2005), Zhu’s doctoral thesis, where the algorithm was given in its modified version. We show the difference in Appendix B. As most research seems to use the version by Zhu et al. (2005), we chose to use this version as well.

### 2.4.5. LabelSpreading

---

**Algorithm 2:** LabelSpreading (Zhou et al. 2004)

---

Compute weight matrix  $W$   
 for  $i \neq j$  (and  $W_{ii} \leftarrow 0$ ) Initialize  $Y^{(0)} \leftarrow (y_i, \dots, y_l, 0, 0, \dots, 0)$   
 Compute the (weighted) diagonal degree matrix  $D$  by  $D_{ii} \leftarrow \sum_j W_{ij}$   
 Compute the matrix  $S \leftarrow D^{-1/2} W D^{-1/2}$   
 Choose a parameter  $\alpha \in (0, 1)$   
 Iterate  $Y^{(t+1)} \leftarrow \alpha S Y^{(t)} + (1 - \alpha) Y^{(0)}$  until convergence  
 Label point  $x_i$  with the class that resembles the index of the highest value in  $Y_i^{(\infty)}$

---

LabelSpreading<sup>9</sup>, outlined in algorithm 2, is quite similar to LabelPropagation. As in algorithm 1, each node  $i$  receives a contribution from its neighbors  $j$ . But there are two differences:

1. Instead of resetting the pre-labeled nodes to their initial state, they only receive an amount of information from their initial state. The relative amount of how much information nodes should receive from neighbors and from the initial label information, is specified via the hyper-parameter  $\alpha$ . This results in a less strong flow from the initially labeled nodes and can even lead to them being re-labeled in the end.
2. The transition matrix  $S$  is constructed differently. Instead of row-normalizing  $W$  like in Zhu et al. (2005) or column-normalizing  $W$  like in Zhu and Ghahramani (2002), it is *symmetrically normalized* with  $D^{-1/2}$ .<sup>10</sup>

LabelSpreading was designed to be a *smooth* function, i.e. the label distribution should not change too much between nearby points. Even if two nearby points initially received two different labels, their difference will be relatively smoothed out during the iterations, depending on the parameter  $\alpha$ . The higher  $\alpha$  is, the less impact the initial labels will have and the more smoothing will happen.

**Example** Figure 2.9 shows the LabelSpreading transition matrix for graph  $G$  from Figure 2.7 and converged label distributions for selected  $\alpha$  values. We can see that  $\alpha$  has great influence on the resulting label distributions. An individual value  $S_{ij}$  in the transition matrix is calculated by multiplying  $1/\sqrt{\text{degree}}$  of nodes  $i$  and  $j$ . From the transition matrix, we can see that node 1 (with label  $A$ ) has greater direct influence on node 3 than node 2 (with label  $B$ ). Therefore, for small  $\alpha$ , node 3 receives a higher confidence score in label  $B$ . Subsequently, its neighboring node 4 which has no other

---

<sup>9</sup>The name was not used in the initial paper, but was mentioned in Bengio et al. 2006 with the reason that the algorithm is inspired by “spreading activation networks”.

<sup>10</sup>Like  $D^{-1}$ , this can be calculated in a computationally easy way by  $D_{ii}^{-1/2} \leftarrow (D_{ii})^{-1/2}$  (all other entries stay 0).

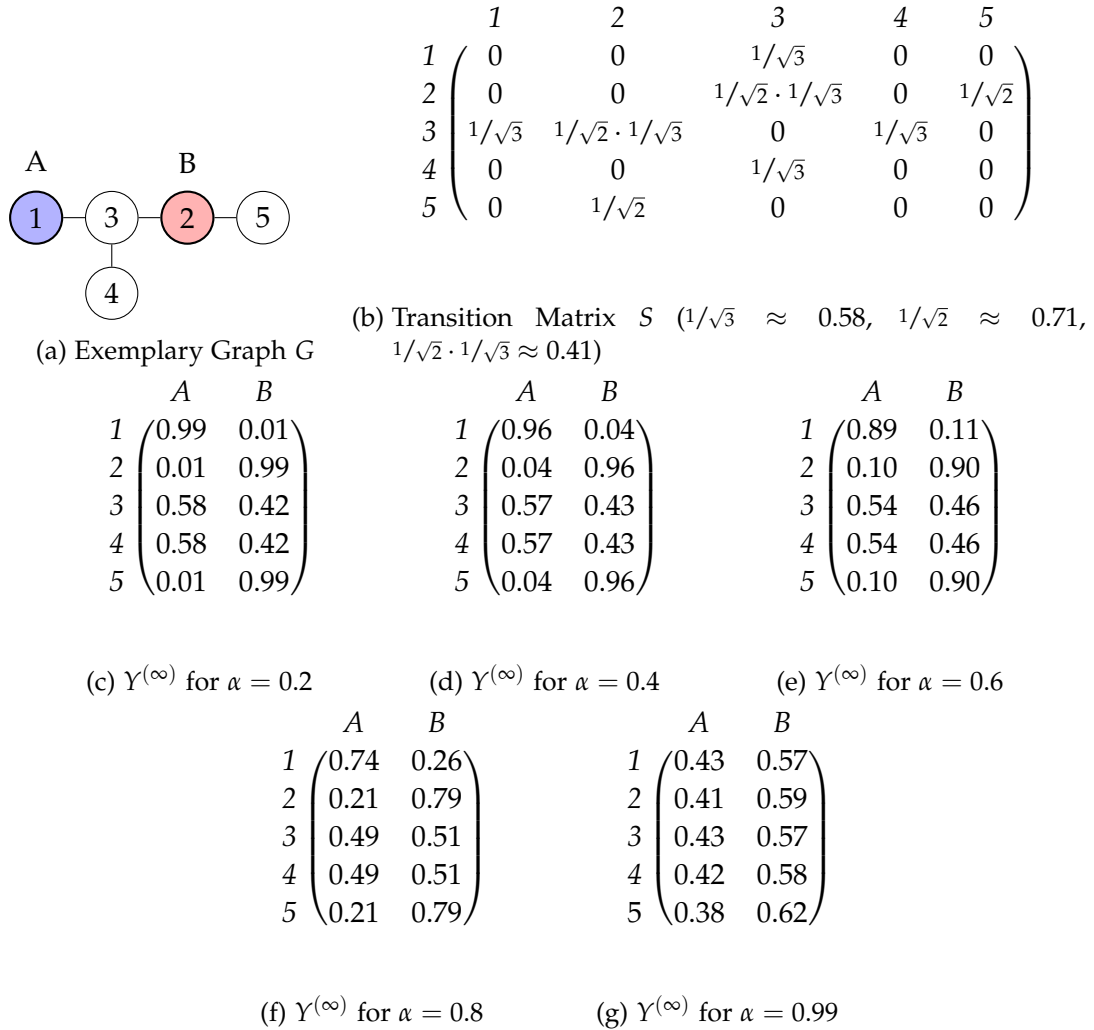


Figure 2.9.: *LabelSpreading* transition matrix for graph  $G$  and selected converged label distributions, depending on  $\alpha$ . All values are rounded to two decimal places. All  $Y^{(\infty)}$  are row-normalized to maintain the interpretation as class probability. We can see that, the higher the hyper-parameter  $\alpha$ , the smoother the overall label distribution across the nodes.

neighbors, receive a higher confidence score in label  $A$  as well. But, for higher  $\alpha$ , the overall influence of label  $A$  decreases. The sole source for this label, node 1, has only a  $1/\sqrt{3} = 0.58$  output, whereas the source for label  $B$ , node 2, has a combined output of  $1/\sqrt{2} \cdot 1/\sqrt{3} + 1/\sqrt{2} = 1.12$ . Because the initial label information are not “refilled” as much as before, label  $B$  gains high influence across all nodes, even on node 1.



**Transition Matrix  $S$  as a Laplacian Matrix** In papers like Bengio et al. (2006) and Buchnik and Cohen (2017), the transition matrix  $S \leftarrow D^{-1/2}WD^{-1/2}$  is described as “normalized graph Laplacian  $L$ ”. This is not the case - according to von Luxburg (2007), the symmetrically normalized graph Laplacian is given by  $L \leftarrow I - D^{-1/2}WD^{-1/2}$ . Rather, and also the way Zhou et al. (2004) called it in their original paper, we have a “symmetrically normalized weight matrix”.

### 2.4.6. Applications

According to Ravi and Diao (2015), label propagation algorithms have been successfully applied to a variety of semi-supervised learning tasks - e.g. computer vision, information retrieval and social networks, as well as natural language processing. Here, we will sketch a few of these applications. Often, modified versions of the basic algorithms described in section 2.4 are used, e.g. to scale them to even larger datasets. The basic idea to propagate labels across a graph structure is the common characteristic.

**Classical Semi-Supervised Tasks** Both papers on the basic label propagation algorithms that we presented in Section 2.4 apply it to a classical semi-supervised problem: Handwritten digit recognition. A demonstration for their performance including code can be found in the scikit-learn library.<sup>11</sup> (Pedregosa et al. 2011) Zhou et al. (2004) additionally demonstrated their *LabelSpreading* algorithm on a text classification dataset, where words were successfully classified into one of the categories *autos*, *motorcycles*, *baseball*, *hockey*.

**Sentiment Lexicon Generation** An interesting application of label propagation is the automatic generation of a sentiment lexicon (Tai and Kao 2013). It is especially interesting because they directly used the *LabelSpreading* approach by Zhou et al. (2004).

**Large-scale deployments at Google & Facebook** Both, Google and Facebook apply label propagation algorithms in various ways with focus on scalability and performance. In his blog post<sup>12</sup>, Ravi from Google describes the use of graph-based machine learning at Google and highlights their *Expander* framework (Ravi and Diao 2015) that is used for solving large-scale problems. These problems require billions of nodes, trillions of edges involving billions of different label types. Examples are the smart response generation in *Inbox by Gmail* (Kannan et al. 2016), automatic categorization of machine generated email (Wendt et al. 2016), translation and visual object recognition. Furthermore, Baluja

---

<sup>11</sup>[http://scikit-learn.org/stable/auto\\_examples/semi\\_supervised/plot\\_label\\_propagation\\_digits.html](http://scikit-learn.org/stable/auto_examples/semi_supervised/plot_label_propagation_digits.html), visited on Nov. 3, 2018

<sup>12</sup><http://ai.googleblog.com/2016/10/graph-powered-machine-learning-at-google.html>, visited on Nov. 3, 2018

et al. (2008) evaluate label propagation for personalized video suggestions on YouTube. Facebook’s Ugander and Backstrom (2013) describe how their algorithm *balanced label propagation* is used to partition the “People You May Know” as it is too big to fit into a single machine.<sup>13</sup> This partitioning is important as the friend suggestions for one user should lie mostly on the same machine, because queries across multiple machines have poor performance.

---

<sup>13</sup>This technique is called *sharding*.

## 3. Implementation

In Section 1.2, we presented an intuitive understanding of combining word embeddings and label propagation to extend a thesaurus. By applying the Pipes and Filter architectural pattern (Buschmann et al. 1996), we modeled this thesaurus extension process as a pipeline and implemented it in Python. The pipeline covers the process from end to end: As input, it starts with the text corpus and the existing thesaurus. As output, it returns a list of predictions in the format of “(word, synset id, confidence)” tuples. It also includes the capability to split the thesaurus in training and test data and automatically calculate performance metrics. The pipeline is published on GitHub.<sup>1</sup>

We first focus on the architectural decisions and the general architecture. Then, we describe each pipeline step and outline the different options we considered and implemented.

### 3.1. Pipeline Architecture

**Pipes and Filters Pattern** The Pipes and Filters architectural pattern breaks the task of a system down into several sequential processing steps (Buschmann et al. 1996). The steps are connected by the data flow through the system - the output data of a step is the input to the subsequent step. Each processing step is implemented by a *Filter* component. The input to the system is provided by *Data Sources*. The output flows into a *Data Sink*, such as a file. The data source, the filters and the data sink are connected sequentially by *Pipes*. The sequence of filters combined by pipes is called a *Processing Pipeline*. The pipes and filters pattern fits the thesaurus extension process well as it is data-centric. We have data sources - the text corpus and the existing thesaurus. Their data gets passed through multiple different transformation steps until the results (prediction tuples and performance metrics) get stored in a data sink (text file). Each of the transformation steps is clearly defined and communicates with the others solely through input and output data.

**Architecture Overview** The pipeline architecture is shown in Figure 3.1. The first step is the pre-processing of the text corpus - the large set of text files with meta-information in each file is transformed to a single file that includes every word, separated by spaces,

---

<sup>1</sup><https://github.com/sebischair/ThesaurusLabelPropagation>

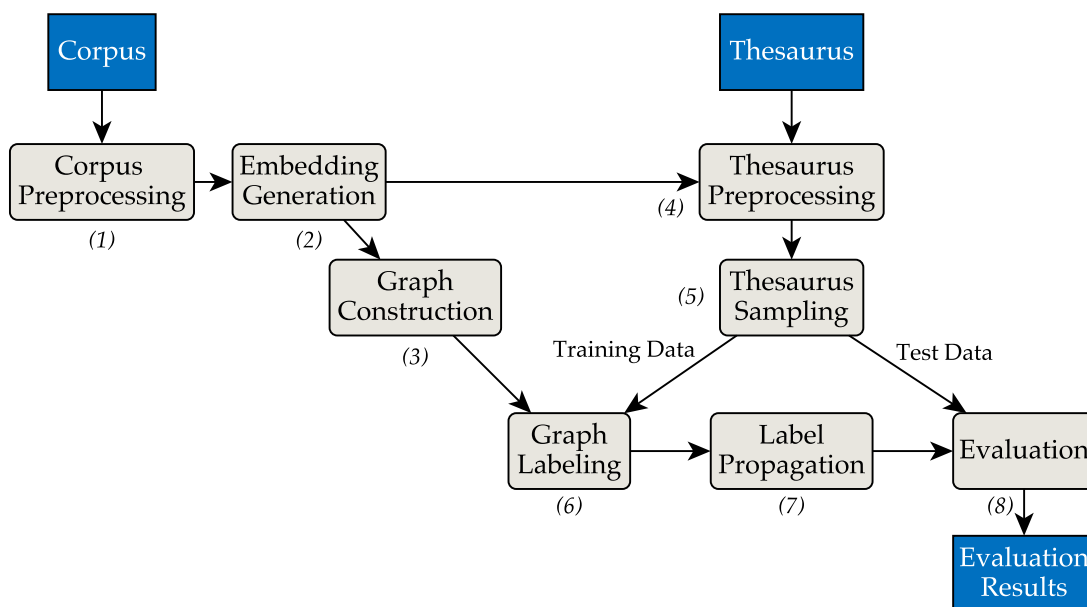


Figure 3.1.: Pipeline architecture that splits the problem into several, successive steps and allows for adjustments in single steps without affecting others.

without punctuation. Then, each word is mapped to a word embedding vector. In the third phase, we use the word embeddings to construct a graph. Words become graph nodes and similar words are connected by edges. As fourth phase, the thesaurus gets pre-processed - from a list of already existing synsets we generate a list of “(word, synset id)” tuples. Words which are not present in the pre-processed corpus are filtered out from the thesaurus. Phase five splits each thesaurus pair into a training and a test set part, phase six labels the similarity graph with the training labels. Phase seven is the propagation phase in which labels get iteratively propagated throughout the graph. It returns a label prediction for each word that got covered by the propagation. The eighth phase calculates relevant performance metrics of the resulting predictions, as e.g. prediction accuracy. It returns the metrics together with the prediction tuples.

For most of the filters, specific hyper-parameters need to be set, e.g. the number of word vector dimensions or the label propagation variant. In general, a hyper-parameter is a parameter whose value is set before the learning process begins. The choice of hyper-parameters for a phase influences its output and therefore has impact on the performance of the final model. Parameter studies to choose proper hyper-parameter values are needed. This is subject to the Quantitative Evaluation in Chapter 4.

**Architectural Goals** During the implementation of the pipeline, we had several goals we wanted to achieve:

1. Hyper-parameters for all filters should be easily accessible and modifiable for each run to make parameter studies easier.
2. A run should be able to use intermediate results from previous runs, so that a re-calculation is not needed.
3. Runs should be reproducible.

We achieved (1) by grouping all hyper-parameter into a configuration file that gets required by the filters. All hyper-parameter values receive a default value, but this default value can be overwritten manually when starting a pipeline run. For (2), we implemented a simple caching mechanism. We not only persist the result of the evaluation phase, but also all intermediate results. At the start of each phase, it is checked whether there already exists output data for the same input and same hyper-parameter configuration. Resource-consuming phases like the calculation of word embeddings or the graph construction, that usually take hours, can therefore often be skipped. (3) is achieved in two respects: For each run, we store the hyper-parameter configuration that led to the run's result. Also, we expose the random number seed as a hyper-parameter. This makes it possible to accomplish deterministic behavior for all our phases, except for the word embeddings phase, where it is almost impossible to achieve fully deterministic results because of multi-threading.<sup>2</sup>

## 3.2. Pipeline Filters

### 3.2.1. Corpus Pre-Processing

**Input and Output** In the corpus pre-processing filter, we transform the texts from the text corpus into a space-separated list of words and join the contents into a single file.

The word embedding algorithms in the next filter will treat every character sequence that is separated by one or multiple spaces as a word and generate an embedding vector for it. For example, a sequence `This is Tom's cat.` will lead to `Tom's` and `cat.` being treated as words, and subsequently receive word embeddings different to `Tom`, `tom`, `cat` or `CAT`. This is why punctuation and capitalization need to be removed.

Then, it can make sense to handle different spellings of the same word that still should lead to the same vector. Moreover, removing special characters like numbers can be useful because e.g. references to specific paragraphs lead to different paragraphs, depending on the referring document, and thus have different semantic meaning.

We apply to all text files as a default the pre-processing actions from Landthaler et al. (2017). Figure 3.2 shows an excerpt of a sample document before and after this pre-processing. The steps are as follows:

---

<sup>2</sup><https://github.com/RaRe-Technologies/gensim/issues/744>, visited on Nov. 3, 2018

<pre>[...] wenn sie im Bundesgebiet Vermögen besitzt, als unbeschränkt steuerpflichtige Kapitalgesellschaft zu behandeln. KStG 1951 §1 Abs. 1 Ziff. 1, §2 Abs. 2 [...]</pre>	<pre>[...] wenn sie im bundesgebiet vermögen besitzt als unbeschränkt steuerpflichtige kapitalgesellschaft zu behandeln kstg PARAGRAPHSIGN abs ziff PARAGRAPHSIGN abs [...]"</pre>
(a) Before	(b) After

Figure 3.2.: Demonstration of pre-processing on a simplified text from the corpus

1. Discard all meta-data like author and document type
2. Remove newline and carriage return characters `\n` and `\r`
3. Replace *muß* with *muss*, the first one corresponds to the old German spelling rules
4. Replace the paragraph sign § with the word *PARAGRAPHSIGN*, followed by a space, so that e.g. “§4d” becomes two words: *PARAGRAPHSIGN* and *4d*
5. Replace every non-alphabetic character (excl. German umlauts and *ß*) with a space
6. Discard words with less than two characters
7. Transform every word to lowercase
8. Save all words, separated by spaces, into a new “cleaned” file

Finally, all individual “cleaned” documents get merged into a single line within a single file. After this pre-processing, we end up with a vocabulary size of 540,025 and a total of 145,091,338 words.

**Variations** We implemented variations to parts of these pre-processing steps and exposed them as hyper-parameters choices:

1. **ß-handling.** Instead of just replacing the *ß* in *muß*, we replaced *every* “ß” with “ss”. Word sense does not change and it prevents other occurrences where two ways of writing (one with *ß*, one with *ss*) are possible. We also implemented a variant where *ß* is never replaced.
2. **Keep characters.** The corpus contains some words from other languages, e.g. French, English and Russian. Instead of just keeping German letters, we kept all alphabetic characters. With that, we prevent that words like *société* get split into “*soci t*” (or rather *soci*, as we discard words with less than two characters). Furthermore, we noticed that many words contain a hyphen, e.g. “portfolio-kapitalanleger”. By default, the hyphen is replaced with a space and the word is split into two words. By keeping hyphens that are located between two characters, we keep the word as one.<sup>3</sup>

---

<sup>3</sup>A different approach, but out of scope for this work, would be the detection of *phrases*, words that

3. **Text Saving.** Instead of merging all documents into a single large line, we put each document into its own line in the output file. This has consequences for the word embedding filter: A newline character resets the history during the training, which means that words in different lines will not be counted as context for each other. If all texts get stitched together as one line, words from the end of an article *A* will be seen as context for words in the beginning of article *B* and vice-versa, but are in fact unrelated contexts.

Table 3.1 shows the resulting hyper-parameters with their respective options, Table 3.2 shows the effects of the  $\beta$ -handling and the “Keep characters” options on the vocabulary size and total word count. We note that especially keeping hyphens has a great effect on the vocabulary size - it increases over 10%.

Parameters	Options		
	$\beta$ -handling	muß → muss	all ß → ss
Keep characters	German letters	Letters	Letters + Hyphens
Text Saving	One line for all texts	One line per text	

Table 3.1.: Corpus Pre-Processing: Hyper-Parameters and possible values

### 3.2.2. Embeddings Generation

**Input and Output** This filter generates a list of multi-dimensional word embeddings from a space-separated list of words.

We set up the three different word embedding technologies that we have explained in Section 2.2.1: word2vec, fastText and GloVe. For all of them, there exist libraries that we could make use of. For word2vec and fastText, the popular *gensim* library<sup>4</sup> by Řehůřek and Sojka (2010) offers a Python interface that we could directly use. For

	Voc. Size		Voc. Size	Total Words
<i>muß</i> → muss	540,025	<i>German Letters</i>	540,025	145,091,338
<i>all ß</i> → ss	536,978	<i>Letters</i>	540,420	145,090,279
<i>no ß</i> → ss	540,026	<i>Letters + Hyphens</i>	609,283	144,309,754

- (a) Effect of  $\beta$ -handling option on vocabulary size (while keeping only German letters)      (b) Effect of “Keep characters” option on vocabulary size and total word count (while setting *muß* → *muss*)

Table 3.2.: Effects of  $\beta$ -handling and “Keep characters” options

very often go together, and generating specific vectors for them. This has the advantage that phrases that get written sometimes written with a hyphen in between, and sometimes not, lead to the same vector.

<sup>4</sup><https://github.com/RaRe-Technologies/gensim>, visited on Nov. 3, 2018

Parameters	Options		
<i>Technology</i>	word2vec	fastText	GloVe
<i>Vector Size</i>	$n \in \mathbb{N}$		
<i>Iteration Number</i>	$n \in \mathbb{N}$		

Table 3.3.: Embeddings Generation: Hyper-Parameters and possible values

GloVe, no popular Python implementation exists<sup>5</sup>, so we had to use the C reference implementation by Pennington et al. (2014) available on GitHub<sup>6</sup> and call the binary files manually from Python.

**Hyper-Parameters** All of the approaches offer multiple hyper-parameters that can be set and that influence the overall performance. As there are whole papers dedicated to parameter studies, we decided to narrow down the parameters we wanted to study. Another reason is that, while word2vec and fastText have almost the same set of hyper-parameters, as this is basically the same approach with the difference in subword (n-grams) handling, GloVe does not offer the same set of hyper-parameters and it would have been necessary to study them separately. For most of the parameters, we therefore settled with the default values. Most importantly, for all approaches, we set the context window size and the minimum count per word<sup>7</sup> to 5. For both word2vec and fastText, we use the CBOW algorithm. Besides the technologies themselves, we will investigate these hyper-parameters during evaluation: *vector size* (dimensionality of the vectors) and *number of iterations*. The hyper-parameters are shown in Table 3.3. After successful embedding generation, we get a list of around 180,000 unique words with their word vectors. The exact number depends on the corpus pre-processing method: 178,446 when just keeping German letters, 188,031 when keeping all letters and hyphens.<sup>8</sup>

As word similarity is calculated from the *cosine similarity* between two word vectors, the vectors' magnitude does not have an effect on the similarity. Nevertheless, the vector have different magnitudes, mostly depending on the frequency a word occurred in the corpus (high occurrence generally leads to higher magnitude). For performance reasons, our approach of calculating similarity does not depend on the angle or the cosine distance, but on the euclidean distance (more on that in the next section). With euclidean distance, the magnitude unwantedly *has* an effect on the distance. We circumvent that by norming all vector to the unit length of 1 before saving.

---

<sup>5</sup>The most popular one, `glove-python`, with around 800 stars on GitHub (visited on Sep. 19, 2018), describes itself as a "toy implementation" and was lastly updated in May 2016. Therefore, it was not selected.

<sup>6</sup><https://github.com/stanfordnlp/GloVe>, visited on Nov. 3, 2018. We used the version with latest commit 07d59d5 from Jun. 24, 2018.

<sup>7</sup>If a word occurs less times than the minimum count, it will be removed.

<sup>8</sup>For both cases, the ß-handling is "muß → muss".



Parameters	Options	
Graph Type	$k$ -nearest-neighbors	$\epsilon$ -neighborhood
Neighbors (for knn-graph)	$k \in \mathbb{N}$	
$\epsilon$ (radius for $\epsilon$ -graph)	$\epsilon \in (0, 2)$	
Edge Weights	unweighted	weighted
Edge Type (for knn-graph)	directed	undirected
Self-References	Yes/No	

Table 3.4.: Graph Construction: Hyper-Parameters and possible values

### 3.2.3. Graph Construction

**Input and Output** This filter generates a similarity graph out of a list of word embeddings and stores its adjacency matrix. Each embedding acts as a graph node.

We examined the two neighbor graph approaches that we explained in Section 2.3: The  $k$ -nearest-neighbor (knn) graph and the  $\epsilon$ -neighborhood graph. For implementation, we used the scientific library “scikit-learn”<sup>9</sup> by Pedregosa et al. (2011), that offers implementations for both approaches.

**Graph-Method specific Hyper-Parameters** For the knn variant, we need to set the  $k$  hyper-parameter. It specifies to how many of its nearest neighbors a node needs to be connected via an edge. Additionally, we need to set whether the edges are undirected, i.e. the graph adjacency matrix should be made symmetric. Note that a knn graph is directed by default, as a node  $x$  can have a node  $y$  as one of its  $k$  nearest neighbors, while the reverse does not hold (there could be a node  $z$  that is closer to  $y$  than  $x$ ). An  $\epsilon$ -graph is undirected by definition. For the  $\epsilon$  variant, we need to set  $\epsilon$  hyper-parameter. Each node gets connected to all other nodes within this neighborhood distance.

**Common Hyper-Parameters** Two other hyper-parameters are relevant for both variants. First, we are interested in the edge weights - how much of a difference does it make whether the edges are unweighted (edges have uniform weight 1) or weighted (the higher the similarity, the higher the edge weight)? Second, we need to set whether self-referencing edges are allowed or not. All hyper-parameters are shown in Table 3.4.

**Distance Metric** As mentioned in Section 3.2.2, we used the euclidean distance for calculating vector neighborhood instead of the usual cosine distance. This is because scikit-learn is not optimized for calculating pairwise cosine distances for such a large number of vectors. Graph-calculation using cosine distances resulted in memory errors,

<sup>9</sup><https://github.com/scikit-learn/scikit-learn>, visited on Nov. 3, 2018

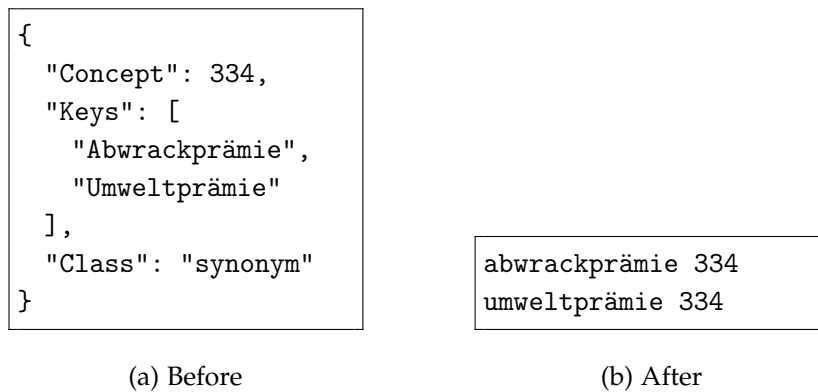


Figure 3.3.: Pre-Processing Demonstration on single thesaurus synset

while calculating their euclidean distances was not an issue. In appendix A, we show that the cosine distance is directly related to the euclidean distance on normalized vectors. The euclidean distance on normalized vectors provides the same neighborhood ordering and can be used to calculate the cosine distance with this formula:

$$\cos_{dist}(A, B) = \frac{\text{euc}_{dist}(A, B)^2}{2}$$

For weighted graphs, we receive the edges with euclidean distance weights from scikit-learn, and then convert it to cosine similarity, which is  $1 - \cos_{dist}$ , where  $\cos_{dist}$  can be calculated with the mentioned formula. Note: If we had set that no self-loops should be allowed, we afterwards need to set the diagonal of the adjacency matrix to 0, as otherwise each node will have a maximum-similarity edge with itself. One additionally has to pay attention to the fact that the  $\epsilon$  hyper-parameter values, as they get used by the scikit-learn graph construction algorithm, will specify the euclidean distance, not the cosine distance.

### 3.2.4. Thesaurus Pre-Processing

**Input and Output** From a thesaurus file with multiple concepts in different word relations and a list of word embeddings, this filter generates a list of  $(\text{word}, \text{synset id})$  tuples.

**Process** First, we discard the concepts that do not describe synonym relations. We define the further thesaurus pre-processing steps as:

1. Transform every word to lowercase.
2. Remove all keys that contain a space. Keys in a synset are sometimes not individual words, but phrases like “tax heaven”. By default, word embedding approaches generate word vectors on a word-level, not on a phrase level.

Parameters	Options	
<i>Remove words with Hyphens</i>	Yes/No	
<i>β-handling</i>	all β → ss	no β → ss

Table 3.5.: Thesaurus Pre-Processing: Hyper-Parameters and possible values

3. Like in the corpus pre-processing in Section 3.2.1, we make each of the two following steps optional via *hyper-parameters* (also shown in Table 3.5).
  - a) Remove all keys that contain a hyphen.
  - b) Replace all  $\beta$  by *ss*.
4. Create an ( $n : 1$ ) mapping from words to synset ids. Note that some words occur in multiple synonym groups - as we restricted ourselves to single-label classification, we had to decide on which synset the word should be mapped to. We solved this decision problem by mapping a word to the group with the most keys.
5. Some words in the thesaurus may not have a corresponding corpus word vector. They will not have a corresponding node in the graph and can not be considered in the label propagation. Therefore, we removed them from the list.
6. In the end, we remove synsets that contain less than two words, as we cannot split them in training and test data later.

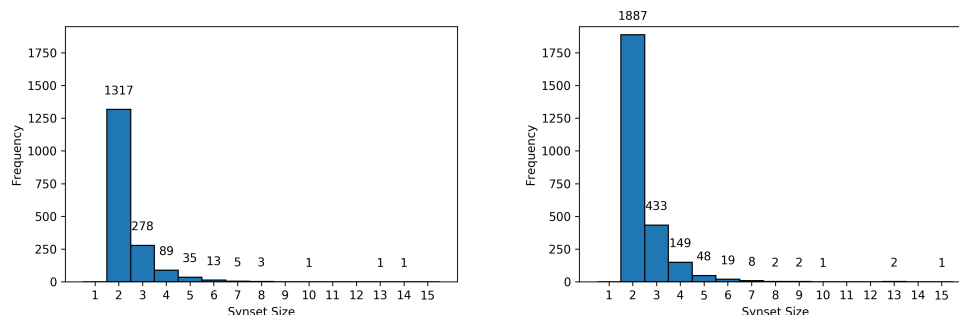
Table 3.6 shows how many words and synsets are left in the thesaurus after each step. The number of synsets and words gets reduced significantly compared to their initial number, especially when removing words with hyphens, too. Figure 3.4 shows the distribution of synsets sizes after pre-processing. When we compare this to Figure 2.2 in Section 2.1.2, we can see that especially the share of synsets with size 3 and 4 is smaller than before.

<i>description</i>	<i>synsets</i>	<i>keys</i>
initially and after 1	12,288	36,076
after 2	12,168	30,014
after 3	11,543	20,709
after 4	11,376 (12,049)	20,366 (29,562)
after 5	5,808 (6,215)	8,238 (9,827)
after 6	1,743 (2,552)	4,173 (6,164)

Table 3.6.: Filtering out synsets and keys from thesaurus dataset in each preprocessing stage. Empty synsets are removed after each step. The number in parenthesis corresponds to the value if phase 3 (removal of keys with hyphens) is omitted.

**Discussion on ( $n : 1$ ) mapping** One could argue that removing words that do not appear in the corpus should have happened before creating the  $n : 1$  mapping. It could happen that, when a word has multiple labels, it gets labeled with a set that disappears

### 3. Implementation



(a) Incl. removal of words with hyphens (b) Without removal of words with hyphens

Figure 3.4.: Histogram on the distribution of synset sizes after fully pre-processing the thesaurus

due to removal of non-corpus words. If the non-corpus word removal would happen earlier, the word would be guaranteed to be added to a set that will not get discarded. As just 1.5% of thesaurus words have multiple labels, and they are getting mapped to the synset with the largest word count (a removal of such a synset because of too many non-corpus words is unlikely), we consider it neglectable.

#### 3.2.5. Thesaurus Sampling

**Input and Output** This filter splits a list of “(word, synset id)” tuples into two lists, training set and test set.

**Variants** We implement two variants as shown in Table 3.7. Either, each synset gets split into a 50% training set and 50% test set part. In case of uneven synset sizes, the training set size gets rounded to the nearest even value, the test set size to the nearest uneven value. The elements in the training set are later used as initially labeled nodes, the test set is used for automatically evaluating the prediction performance. Or, the whole list gets passed to the training set. This way, as much training information as possible is available, but evaluation has to be conducted manually.

Parameters	Options	
<i>Training/Test Split</i>	50%/50%	100%/0%

Table 3.7.: Thesaurus Sampling: Hyper-Parameters and possible values

To avoid that our model overfits to a specific training-test-set split, we generate multiple split variants and evaluate our models for each of these splits. We generate these split variants by explicitly setting the random number generator seed to a different value for each variant.

Different splits other than splitting each synset in half are not really possible. As seen in Figure 3.4, the vast majority of synsets just has two keys. On the one hand, we need at least one key per synset in the training set, or the respective synset will not get predicted. On the other hand, we need at least one test key per synset as otherwise we could not evaluate if this synset gets correctly predicted.

### 3.2.6. Graph Labeling

**Input and Output** From the training set of “(word, synset id)” tuples and the graph adjacency matrix, this filter labels the respective graph nodes with their synset ids. All words that do not have such an initial training label get marked with  $-1$ .

### 3.2.7. Label Propagation

**Input and Output** From a sparsely-labeled graph, this filter determines labels for the unlabeled nodes. It returns a list of “(word, predicted synset id, confidence, [top3 synset ids])” tuples. The words that were already labeled are included in this list.

We implemented the two basic algorithms described in Section 2.4: *LabelPropagation* and *LabelSpreading*. The implementation took place using the Python frameworks “numpy”<sup>10</sup>, “scipy”<sup>11</sup> and “scikit-learn”, that enable fast operations on large matrices, especially through the sparse matrix object types of scipy.

**Considerations** scikit-learn<sup>12</sup> includes implementations for *LabelPropagation* and *LabelSpreading*. We switched to our own implementation due to these reasons:

- Their implementation considered the Graph Construction (see Section 3.2.3) part of label propagation. This means that the graph has to be re-generated each time the method is executed, even when the underlying graph is the same and just a hyper-parameter of the actual propagation part is changed. By implementing it ourselves, we could introduce caching behavior.
- We recognized that the *LabelSpreading* implementation was not fully in line with the reference paper by Zhou et al. (2004). First, nodes in their knn-graph contained self-loops, which was undesired originally. Second, their calculation of the transition matrix (“Laplacian Matrix”) expected the graph to be symmetric, which is not the case for the default knn-graph that is not explicitly made undirected.<sup>13</sup>

---

<sup>10</sup><https://www.numpy.org>, visited on Nov. 3, 2018

<sup>11</sup><https://www.scipy.org>, visited on Nov. 3, 2018

<sup>12</sup>[http://scikit-learn.org/stable/modules/label\\_propagation.html](http://scikit-learn.org/stable/modules/label_propagation.html), visited on Nov. 3, 2018

<sup>13</sup>We notified the project’s maintainers of these inconsistencies here: <https://github.com/scikit-learn/scikit-learn/issues/11784>, visited on Nov. 3, 2018

Parameters	Options	
Type	LabelPropagation	LabelSpreading
Iteration Number	$n \in \mathbb{N}$	
$\alpha$ (for LabelSpreading)	$(0, 1)$	

Table 3.8.: Label Propagation: Hyper-Parameters and possible values

**Hyper-Parameters** The hyper-parameters for this filter are shown in Table 3.8. We treat the number of propagating iterations as a hyper-parameter, similarly to Buchnik and Cohen (2017). Usually, as both algorithms converge (Bengio et al. 2006), iteration is done until convergence. Here, we directly cut off after a number of iterations. LabelPropagation does not have another hyper-parameter, while LabelSpreading needs the smoothing value  $\alpha \in (0, 1)$ .

After the iterations took place, we row-normalize the label distribution matrix  $Y$ . We treat the matrix as a confidence matrix. We can then note the confidence between 0 and 1 (0 – 100%) for each “(word, synset id)” pair. It states the algorithm’s confidence for applying the “synset id” label to the respective word. For each word, the filter returns the label that received the highest confidence. More information like the label’s exact confidence value and the labels with second and third highest confidence are returned as well for evaluation purposes.

### 3.2.8. Evaluation Filter

**Input and Output** From a list of “(word, predicted synset id, confidence, [top3 synset ids])” and the training and test set lists (format: “(word, synset id, confidence)”), this filter computes various quantitative evaluation metrics and stores the input lists as a combined list.

On the one hand, we compare the predictions with the “true” labels in the test set. On the other hand, we calculate metrics on the predictions themselves, such as synset size distribution or mean confidence. The exact evaluation metrics are subject to Chapter 4, *Quantitative Evaluation*. These quantitative results get saved in a statistics file. The three input lists (predictions, training & test set) get saved, too, as a combined file for further manual evaluation.

## 4. Quantitative Evaluation

In Chapter 3, *Implementation*, we identified several hyper-parameters along the pipeline’s phases which can be varied in many different ways. In so-called parameter studies, we try to understand the impact of certain hyper-parameters on the overall performance. We do so by compare the performance of various *configurations* (a specific choice of hyper-parameter values). We calculate the accuracy of the synset predictions on the test set. As the accuracy can be calculated automatically, we can try out many configurations in order to find the best possible set of hyper-parameter values. In the end of this process which we call “Quantitative Evaluation”, we combine the best hyper-parameter values and derive two optimized configurations. In Chapter 4, *Qualitative Evaluation*, we use these optimized configurations to verify if good quantitative performance corresponds to good human rating.

### 4.1. Structure

The quantitative evaluation flow is visualized in Figure 4.1. To automatically assess a run’s performance, we split the existing handcrafted thesaurus into a training and a test set. The training set data is used as initial labels for the similarity graph. Via label propagation, these labels get propagated along the graph so that previously unlabeled nodes receive a label. The test set then is used to calculate the prediction accuracy. Having just one partition into training and test set makes the evaluation vulnerable to over-fitting. We overcome that by controlling the random seed that determines the partition selections. We create three different partitions and evaluate each run’s configuration on all three.

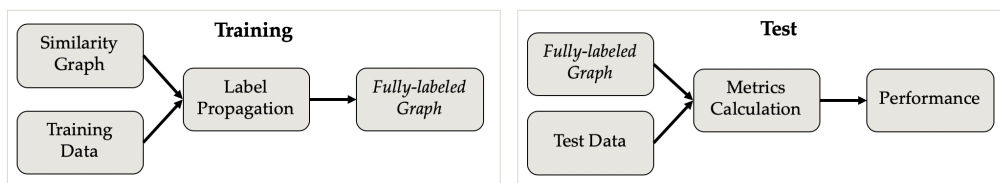


Figure 4.1.: The Quantitative Evaluation as a Training and a Test phase.

**Accuracy** The central metric for our quantitative evaluation is the accuracy, the rate of test samples whose class gets correctly predicted:

$$\frac{\sum_i [y_{pred_i} = y_{test_i}]}{|y_{test}|}$$

In a single-class scenario, accuracy alone would not be enough to get a good estimate on the quality of our prediction. The model could just always predict the more frequent outcome and get an accuracy over 50%. One would have to calculate precision and recall to judge the results. But as we are in a multi-class scenario with many classes, where no class is significantly larger than the others, just predicting the most frequent class will still lead to low accuracy on itself.

## 4.2. Parameter Studies

In the following sections, we show the results of parameter studies around the several phases of our pipeline. We arrange our parameter studies into four groups. We start with the embeddings generation, as the choice of embeddings had the largest impact on performance. We then investigate the graph construction and label propagation phases. In the end, we evaluate choices for text corpus and thesaurus pre-processing.

Pipeline Phase	Modifiable Parameter	Base Configuration Value
<i>Embedding Generation</i>	Technology	fastText, word2vec
	Dimensions	100
	Iterations	5
<i>Graph Construction</i>	Type	k-nearest-neighbors
	Neighbors	3
	$\epsilon$ (radius for $\epsilon$ -graph)	(no default)
	Edge Weights	Unweighted
	Edge Type (for knn-grah)	Undirected
	Self-references	No
<i>Label Propagation</i>	Type	LabelPropagation
	Iteration number	3
	$\alpha$ (for LabelSpreading)	0.2
<i>Pre-Processing</i>	ß-handling	muß → muss
	Keep characters	German letters
	Text Saving	One line for all texts

Table 4.1.: Modifiable parameters per pipeline phase and their default values. These parameters are subject to our parameter studies.



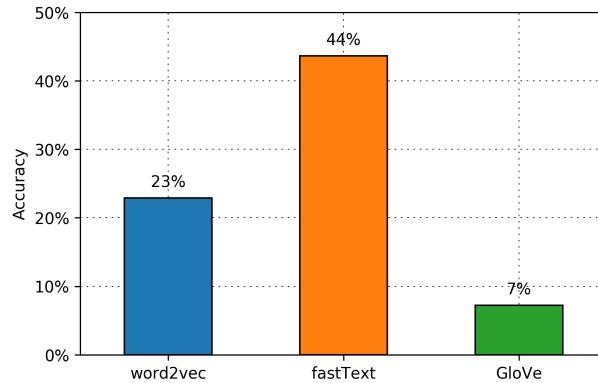


Figure 4.2.: Accuracy of word2vec, fastText and GloVe technologies compared

Via manual experimentation, we have deduced a parameter configuration which we use as default as it is sensible to changes of individual parameters. We fix this base configuration and vary an individual parameter to see how the parameter impacts the overall performance. The modifiable parameters and the base configuration are given in Table 4.1. Note that we perform all parameter studies on fastText and word2vec embeddings because we will deduce parameter configurations for each of these embedding technologies.

#### 4.2.1. Embeddings Generation

**Technology** Figure 4.2 shows the accuracies of the three embedding technologies word2vec, fastText and GloVe with our default parameters. fastText embeddings perform best by far with over 44% accuracy. word2vec’s accuracy is almost half of that with only 23%. With just 7% accuracy, we note that the default GloVe word embeddings perform significantly worse than fastText and word2vec ones. We observed the same behavior when varying the other parameters from Table 4.1. Therefore, we excluded GloVe embeddings from further evaluation.

**Dimensionality and Iteration Number** The accuracy generally slightly improves with higher embedding dimensionality, as can be seen in Figure 4.3 (a). Interestingly, we can see a maximum at 400 dimensions for fastText, while for word2vec, the accuracy slightly decreases compared to 300 dimensions. In Figure 4.3 (b), we see that with higher iteration number, the accuracy improves as well. This is especially true for word2vec where we can see an increase of almost 10 percentage points from 5 to 60 iterations. With fastText, the increase is less than 4 percentage points. A higher iteration number seems to be more important than higher dimensionality. For both parameters, we can see that the accuracy gain is neglectable as long as the values are chosen sufficiently high.

## 4. Quantitative Evaluation

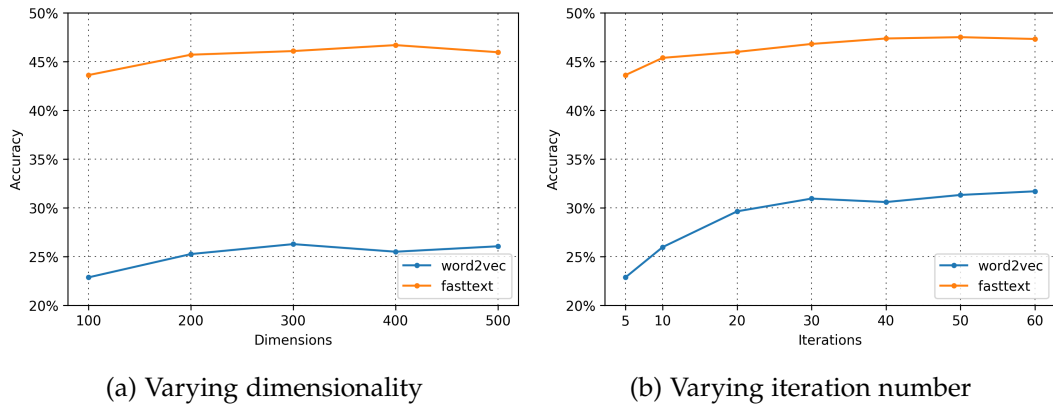


Figure 4.3.: Accuracy with varying dimensionality and iteration number compared.

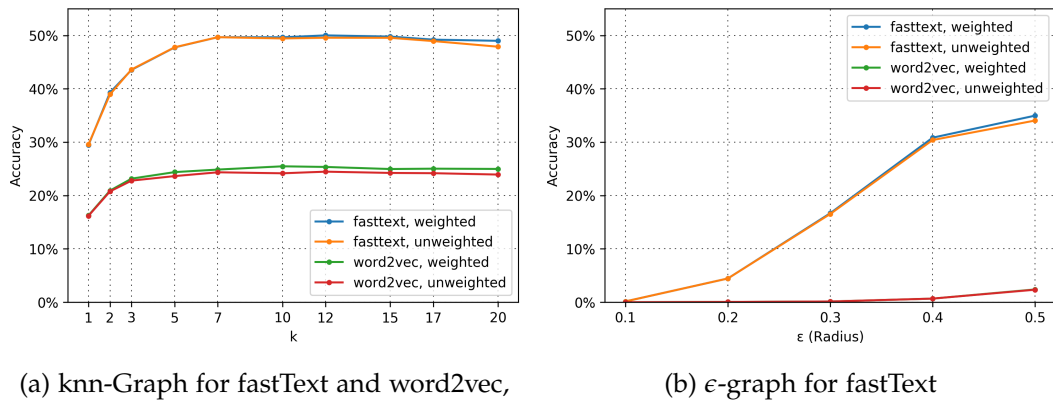


Figure 4.4.: Accuracy for knn-Graph and  $\epsilon$ -graph compared

### 4.2.2. Graph Construction

**Graph Type** Figure 4.4 (a) shows the accuracy for a  $k$ -nearest-neighbors graph with varying  $k$ . Until around  $k = 7$ , increasing  $k$  results in higher accuracy. Interestingly, the increase for fastText is much higher than for word2vec. By increasing  $k$ , an accuracy of 50% can be reached, a value that could not be reached through increasing the embeddings dimensionality or iteration number. For word2vec, high  $k$  achieve the same accuracy levels as high embeddings dimensionality, but not the same as simply increasing the iteration number. Figure 4.4 (b) shows the accuracy for an  $\epsilon$ -neighborhood graph with varying  $\epsilon$ . In general, we can see that the accuracy is far less than when using a  $k$ -nearest-neighbors graph. Larger  $\epsilon$  values do result in higher accuracy, but also in an increase in graph construction time. The construction time for an  $\epsilon$ -graph was multiple times higher than for a  $k$ -nearest-neighbor graph, even when the resulting accuracy was much less. Therefore, we did not further investigate  $\epsilon$ -graphs.

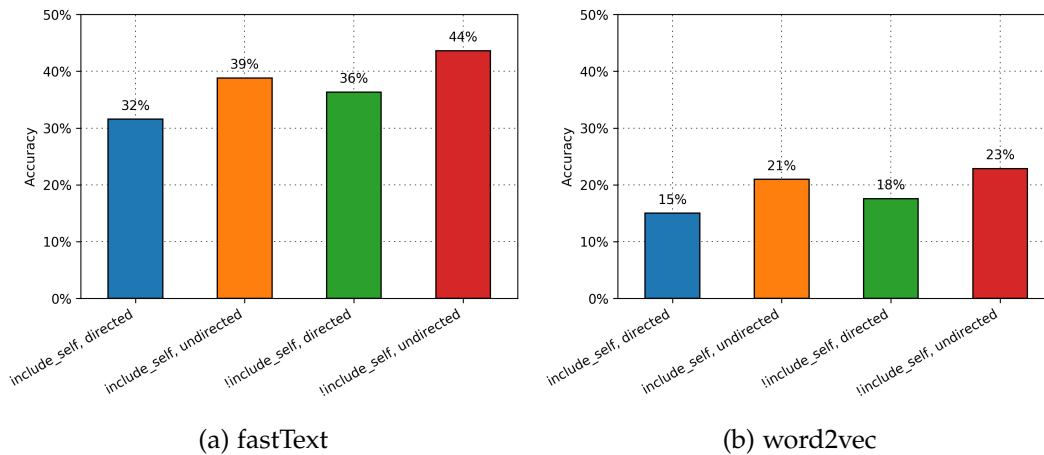


Figure 4.5.: Accuracy for Self-References and Edge Type Parameters Compared

**Edge Weights** Regarding the edge weights, we notice a slight accuracy advantage for weighted edges compared to unweighted edges. That is unexpected, we expected a higher advantage. Edge weights contain information about the similarity of two nodes: The higher the weight, the more similar. An edge in an unweighted graph, however, just signals that nodes are within each other’s neighborhood. From Section 2.4, we know that label propagation algorithms propagate more label information along high-weighted edges. Still, this plus of label information did not lead to significantly better predictions.

**Edge Type and Self-References** Figure 4.5 shows the performance behavior for setting edges directed/undirected and for explicitly allowing/disallowing node self-references. For both fastText and word2vec embeddings, undirected edges result in better result than directed ones. This is interesting: On a weight matrix level, undirected edges correspond to directed edges, with the addition that for each edge, an inversely-directed edge is added if not already existent. One could argue that this inverse edge does not add any value or even degrades performance, as it leads to a node that is not in the direct neighborhood of a node. Our results seem to indicate that the inverse edges do add value. Disallowing self-references results in better performance as well, but does not seem to have as much of an effect. A combination of these two options results in the best performance. Hence, we kept it as a default setting for all future analysis.

### 4.2.3. Propagation Phase

In Figure 4.6, we compare LabelPropagation and LabelSpreading for varying iteration numbers. For LabelSpreading, we used  $\alpha = 0.2$  as default value, as it is done by scikit-

## 4. Quantitative Evaluation

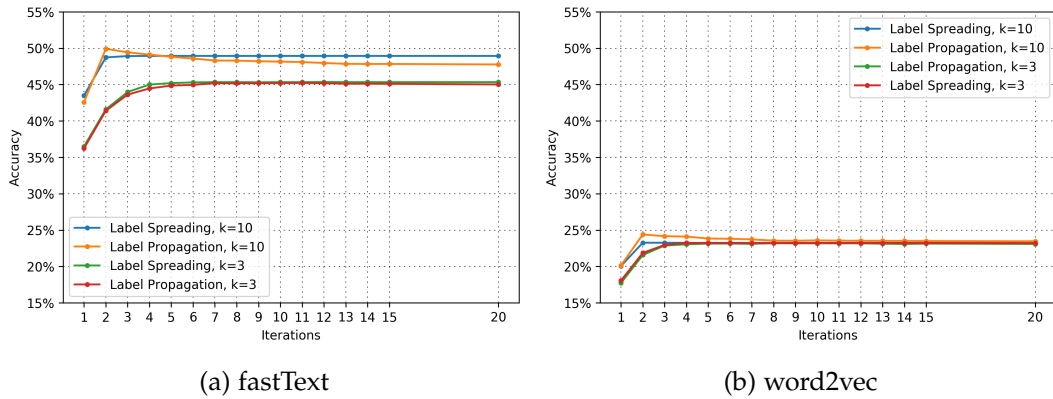
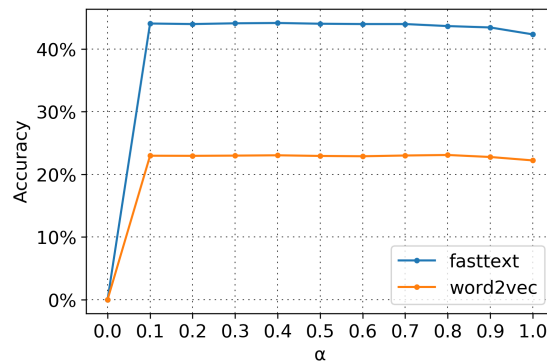


Figure 4.6.: Accuracy with varying label propagation technology & iteration number for two different graph configurations

learn.<sup>1</sup> For a  $k$ -nearest-neighbor graph with  $k = 3$ , we could not determine a difference in accuracy between the two approaches. Therefore, we have generated the results for a graph with  $k = 10$  as well. LabelSpreading behaves similar to  $k = 3$ , although its performance does not change anymore after an initial increase in the first 2 – 3 iterations. LabelPropagation behaves completely different than for the previous  $k = 3$  graph. It reaches the highest performance overall, including all LabelSpreading runs, after 2 iterations. From there, its performance is decreasing until it remains constant after around 12 iterations. In the end, its accuracy is lower than LabelSpreading’s. This applies to fastText embeddings. For word2vec embeddings, after the initial LabelPropagation spike, all variants converge to around the same accuracy. We chose to ignore LabelPropagation’s high performance for low iterations numbers and settle for a higher iteration number where the performance value remains constant. For low iteration numbers, the result is probably very dependent on the initial node neighborhood and could vary a lot when modifying the input from other phases. Also, we chose LabelSpreading as “optimized” configuration method as its accuracy for high iterations is higher than for LabelPropagation.

**The  $\alpha$  parameter in LabelSpreading** Figure 4.7 shows LabelSpreading resulting accuracies for different  $\alpha$  values. Although LabelSpreading is not defined for  $\alpha \in \{0, 1\}$ , we chose to calculate performance for these two values as well as its behavior can be intuitively explained. For  $\alpha = 0$ , in each iteration, no label information from neighboring nodes is applied to the resulting label distribution matrix. Therefore, no nodes get labeled, accuracy is 0%. For  $\alpha = 1$ , the initial labels do not get re-applied to the graph, there is no constant flow from initially labeled nodes anymore. Interestingly, accuracy does not decrease that much at the same time. Overall, we see that the choice of  $\alpha$  does

<sup>1</sup>[http://scikit-learn.org/stable/modules/generated/sklearn.semi\\_supervised.LabelSpreading.html](http://scikit-learn.org/stable/modules/generated/sklearn.semi_supervised.LabelSpreading.html) visited on Nov. 3, 2018

Figure 4.7.: Accuracy for LabelSpreading with varying  $\alpha$ 

not seem to have a large effect on the accuracy as long as the value is not 0. Therefore, we chose to stay with  $\alpha = 0.2$ .

#### 4.2.4. Pre-Processing

Regarding pre-processing, the choice of  $\beta$ -handling and the type of text saving did not have a significant impact on accuracy and is therefore not further explained here. However, the variant of “Character Keeping” has a clear effect - in Figure 4.8, we can see that keeping hyphens has a positive effect on accuracy. This is not surprising, as inclusion of words with hyphens resulted in considerably more synsets and keys left after thesaurus pre-processing and therefore more training data (see Chapter 3, *Implementation* for exact numbers).

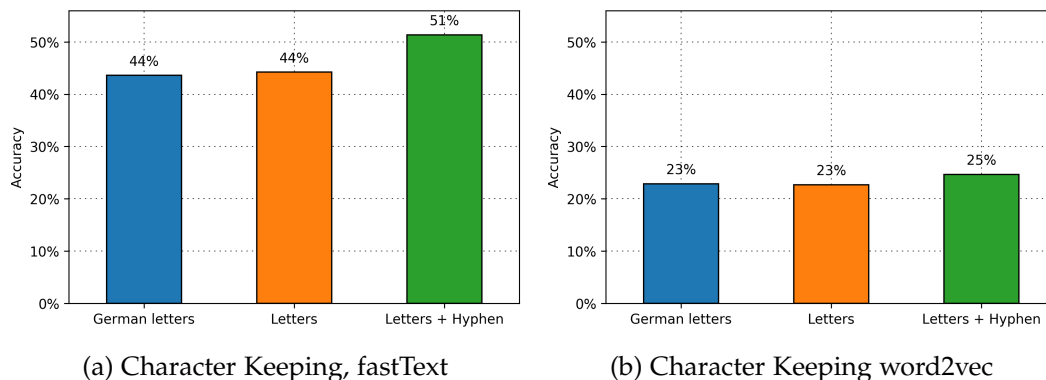


Figure 4.8.: Accuracy with varying pre-processing settings

### 4.3. Optimized Configurations

From the results of our parameter studies, we inferred an optimized configuration and applied it to `fastText` and `word2vec` as embeddings technology:

- **Embeddings Generation.** We saw that a high dimensionality and a high iteration number result in higher accuracy. Therefore, we use an embedding dimensionality of 400 and an iteration number of 40.
- **Graph Construction.** Our analysis showed that the  $k$ -nearest-neighbor graph generally performs better than the  $\epsilon$ -neighborhood graph. We have identified good performance for  $k > 7$ . Therefore, we use a  $k$ -nearest neighbor graph with  $k = 12$ . Furthermore, we have seen that weighted and unweighted edges result in around the same accuracy, with a slight advantage for weighted edges. Therefore, we use weighted edges. Both undirected edges and omitting self-references result in higher accuracy as well and we choose to use these options.
- **Label Propagation.** We found that that `LabelSpreading` performs slightly better than `LabelPropagation` for high iteration numbers. Also, the  $\alpha$  value does not have great impact on the performance, as long as it is not 0 or 1. Therefore, we use `LabelSpreading` with the default value  $\alpha = 0.2$  and with 15 iterations.
- **Pre-Processing.** Our comparisons show that keeping all letters and hyphens results in a clearly better performance than just keeping German letters. Therefore, we use this type of pre-processing and keep the -handling and text saving variants to their defaults (`muß`  $\rightarrow$  `muss`, and saving of all texts into a single line). After embeddings generation, this leads to 188,031 unique words.

	Accuracy	Top3 Accuracy
<i>fastText</i>	62%	79%
<i>word2vec</i>	41%	56%

Table 4.2.: Accuracy and Top3 Accuracy of optimized configurations for `fastText` and `word2vec` embeddings

The resulting accuracies are shown in Table 4.2. The table also shows the “top3 accuracy”, which is the share of the correct class being within the top 3 of the candidate classes. The values are based on training with 3,277 words over 2,552 synsets. The test set contains 2,887 words. Combining the best performing parameters into optimized configurations has a considerable effect on the resulting performance. For `fastText`, now 61% of the predictions on the test set are correct, compared to 51% (by keeping words with hyphens). Same for `word2vec`, where accuracy is now 41% when it was previously around 32% (by choosing a high iteration number during embedding generation). The top3 accuracy behaves accordingly - for both configurations, it is around 15% higher than the accuracy itself.

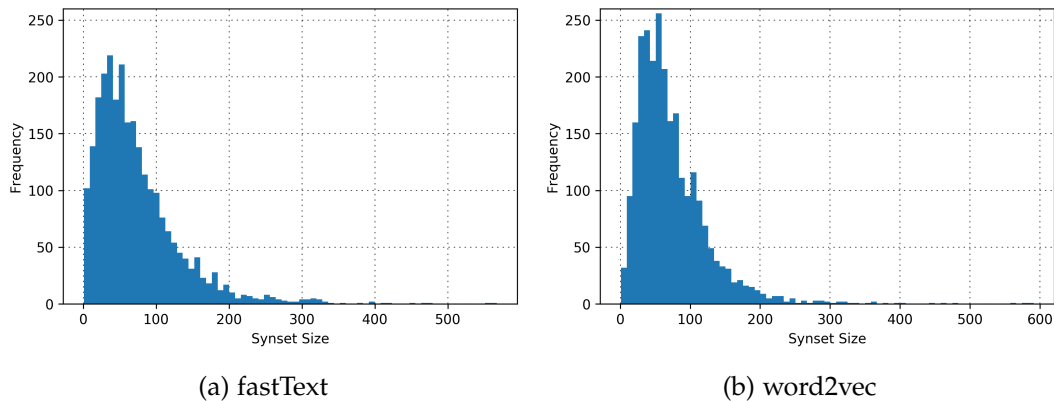


Figure 4.9.: Synset distributions of optimized configurations for fastText and word2vec embeddings

In Figure 4.9, we have visualized the synset distributions of each optimized configuration.<sup>2</sup> Training words are included in this view. For both configuration types, the mean size is 73.7 words per synset. For fastText, all words have received a label, but for word2vec, 14 words were not labeled. While the distributions look similar overall, we can see that with fastText slightly more smaller synsets were produced (median size 58, max size 567), while with word2vec more medium and large synsets were produced (median size 61, max size 591).

An automatic test can just verify whether or not the existing thesaurus' words were classified as intended. In this quantitative evaluation, we assumed that good performance on the existing thesaurus generalizes to good performance over all words of the text corpus. In the next chapter, the manual (qualitative) evaluation, we try to verify this assumption and manually rate the prediction quality of the optimized configurations.

<sup>2</sup>These synset distributions correspond to a specific training/test split, in contrary to accuracy and top3 accuracy before, where the value was averaged over three splits with different random seeds.





## 5. Qualitative Evaluation

We conducted multiple manual evaluations to learn more about the predictions that get produced by label propagation. This is especially important since an automatic evaluation of the prediction quality on words, that were not part of the original thesaurus, is not possible. We evaluated predictions for the two optimized configurations from Chapter 4. In the Quantitative Evaluation, we split the existing thesaurus into training and test set to evaluate the performance on the test set. In the Qualitative Evaluation, all algorithms are trained with the full thesaurus.

Per manual evaluation, a list of 54 synsets was shown to human raters. For each of the synsets, the synset’s training words as well as 10 words that were predicted for it, were listed.<sup>1</sup> In a group of two persons, we manually rated whether a word fit into the suggested synset. An exemplary manually labeled synset is shown in Figure 5.1. We used these three rating levels:

- 0 Not similar to the predicted synset
- 1 Similar to the predicted synset, because the word belongs to the same semantic area. Examples: “kraftfahrzeug” for a synset with concept “dienstwagen”, “zeitungsanzeigen” for a synset with concept “zeitungsausträger”.
- 2 Should be added to the synset.

We needed to develop a process that determines which synsets and which of the synsets’ predictions should be shown to the raters. Therefore, we split our qualitative evaluation into two parts: *Pre-study* and *main study*. With the pre-study, we determine how certain characteristics, that can be used for synset and prediction selection, correlate with high

Existing Synset	Suggestion	Score
15396 zeitungsausträger	1 zeitungsausträgerinnen	2
zeitungsträger	2 zeitungsausträgern	2
zeitungszusteller	3 zeitungszustellern	2
	4 zeitschriftenwerber	1
	5 zeitungsverleger	1
	6 zeitungsanzeigen	1
	7 zeitungsträgern	2
	8 zeitungsboten	2
	9 zeitungsaustragen	2
	10 zeitungsverlagen	1

Figure 5.1.: Screen shot of a manually evaluated synset

<sup>1</sup>Or less, if there were less than 10 predictions for the respective synset

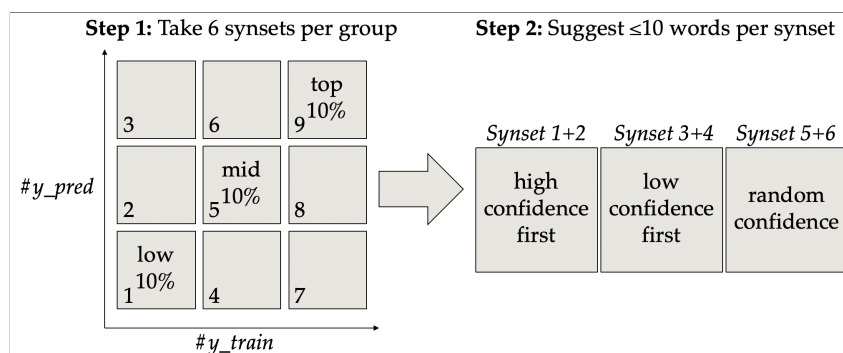


Figure 5.2.: Pre-Study Selection Procedure

human rating. We then made use of those correlations to select suitable candidates for the main studies. Our goal was to show predictions with characteristics that correlate with high rating.

At the end of this chapter, we discuss whether or not the results from the quantitative evaluation could be verified.

## 5.1. Pre-Study

We want to analyze if and how these three characteristics correlate with prediction quality:

- (a) Number of training words in synset ( $\#y_{train}$ )
- (b) Number of suggested words for synset ( $\#y_{pred}$ )
- (c) Confidence

As an example: For a synset with a small number of predictions, the predictions might have a higher quality than the ones for a synset with lots of predictions (or reverse).

The pre-study procedure is shown in Figure 5.2. We group all synsets according to attributes (a) and (b). For example, group 6 groups all synsets with a medium size of training words (between 45% and 55% quantile, inclusive) and with a high number of predicted words (over or equal the 90% quantile). We set the quantile limits *inclusive*. Therefore, a synset can be assigned to multiple groups. The reason for setting the limits inclusive is that often there is not enough variety in the possible values. For example, the 10% quantile of  $\#y_{train}$  is 2, which is also the lowest possible training synset size.

From each of the nine groups, we draw six random synsets, where each synset can be drawn only one time. We order the predictions within each synset by the confidence score. For two of the six synsets, words with high confidence score are shown first. For another two, words with low confidence are shown first. For the last two, a random

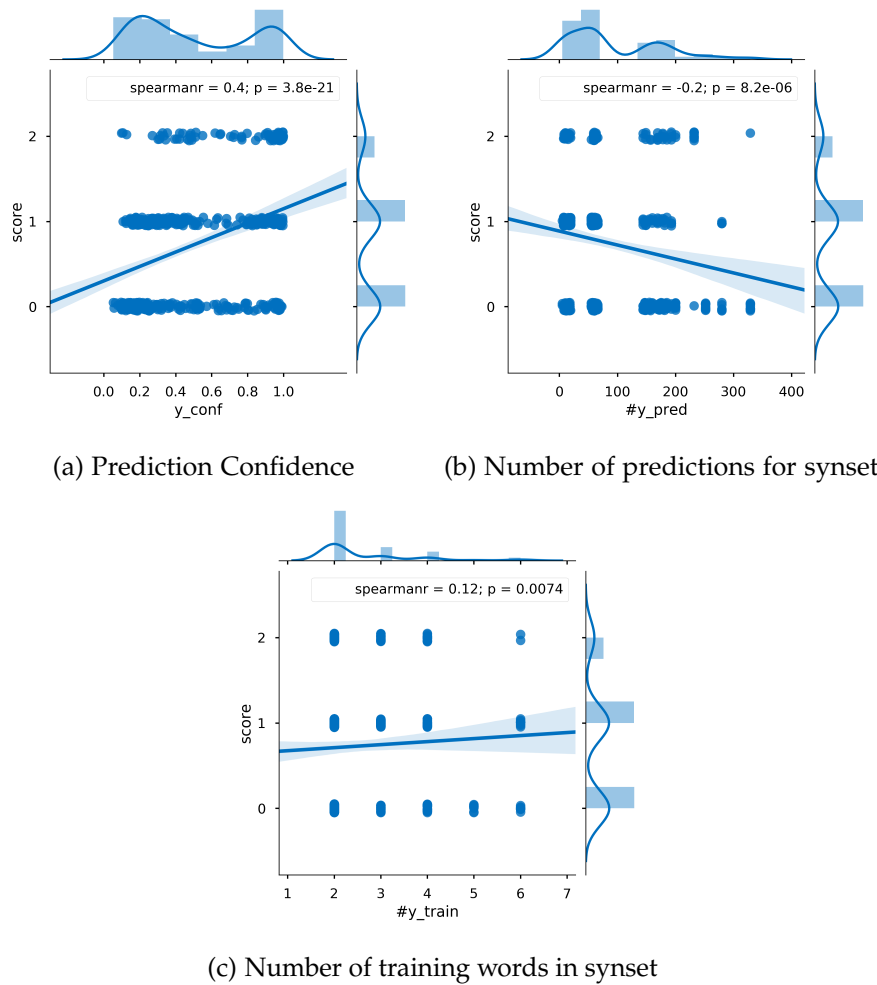


Figure 5.3.: Correlation analysis between high rating and prediction characteristics

order is applied. The first 10 predictions of each of these synsets are then shown to the human evaluators. The characteristics that led to a selection are stored invisibly and are used for calculating the correlations, after the ratings have been submitted.

**Results** We conducted the Pre-Study with the predictions of the optimized propagation configuration with fastText embeddings. For the 54 drawn synsets, we rated an overall of 500 words. 42.4% (212), received a score of 0, 42.0% (210) a score of 1, and 15.6% (78) a score of 2.

The pairwise relationships between the three characteristics and the rating are shown in Figure 5.3. The relationships are visualized as linear regression plots with a 95% confidence interval using the data visualization library seaborn<sup>2</sup>. The actual data is

<sup>2</sup><https://seaborn.pydata.org>, visited on Nov. 3, 2018

shown as scatter plot. The distribution of the respective variables is shown outside the top and right axes. A confidence interval is an estimate that might contain the true value of the parameter with a high probability (Keener 2010). The plots are annotated with the Spearman’s rank correlation coefficients and  $p$ -values for the respective relationship. The Spearman’s rank correlation coefficient describes how well the relationship between two variables  $X$  and  $Y$  can be described by a monotonic function. There is a *positive correlation* when large values of  $X$  have a tendency to be associated with large values of  $Y$  and small values of  $X$  with small values of  $Y$ , and a *negative correlation* if the relationship is vice-versa (“Spearman Rank Correlation Coefficient” 2008). A perfect positive correlation results in a correlation coefficient of  $+1$ , a perfect negative correlation in a coefficient of  $-1$ . When the correlation coefficient is close to  $0$ , the variables are not correlated. We can identify certain correlations:

- *Moderate positive correlation* (spearmanr = 0.4) between the prediction confidence and the rating.
- *Weak negative correlation* (spearmanr =  $-0.2$ ) between the number of predictions for a synset and the rating.
- *Weak positive correlation* (spearmanr = 0.12) between number of training words in a synset and the rating.

Under a significance level of  $\alpha = 0.05$ , all correlations are significant ( $p < \alpha$ ). We conclude from our results that we should filter for synsets with a high number of training words and a low number of predicted words and then order predictions by high confidence. This increases the likelihood for high human ratings.

## 5.2. Main Study

We conducted two main studies and for each we again start with selecting 54 synsets. A synset could be subject to multiple main studies. We chose synsets with a high number of training words (greater or equal 80% quantile) and a low number of predicted words (smaller or equal 20% quantile). The suggestions were ordered by descending confidence value, the top 10 were taken for rating by us.

**Results** The results are shown in Table 5.1.<sup>3</sup> We note that our goal of selecting synsets and predictions that have promising characteristics was successful. The results for fastText are considerably better than they were in the pre-study. Only a 6.5% share of the predicted words is not related at all to the original synset, compared to previously over 40%. Like for the quantitative evaluation, the optimized configuration with fastText embeddings performs better than with word2vec embeddings. For fastText, we receive

---

<sup>3</sup>Note: For word2vec, two persons individually evaluated the predictions. The results were then averaged.

---

Variant	Score (Share in %)		
	0	1	2
<i>Propagation: fastText</i>	6.5	63.9	29.6
<i>Propagation: word2vec</i>	62.2	29.6	8.2

Table 5.1.: Main Study Results, rounded to one decimal

a synonym rating for around 30% of the overall predictions. For wordvec, the result is worse, with over 60% of the predictions not being related to the concept at all. The humanly-rated difference in performance between the two embeddings types seems to be even greater than the difference in the automatic quantitative evaluation. To answer our question, whether or not the qualitative evaluation can verify the results from quantitative evaluation: We could verify the difference in prediction performance between fastText and word2vec embeddings. But, the performance values of the two evaluations do not seem to be really comparable.

Up to now, we have collected performance results for the label propagation approach in an automatic and in a manual fashion. We have not yet studied what these results really mean: How does our approach compare to a simpler baseline approach? What kinds of synsets have to be extended? Are all kinds of synsets extendable in the same way? We will deal with these questions in the next chapter.



## 6. Assessment and Further Refinements

In this chapter, we put our previous results into context. At first, we present a baseline approach with the same objective as our label propagation configurations. This baseline approach operates directly on word embeddings and uses a nearest-neighbor strategy to suggest synset extensions. We compare its performance with the results reached with label propagation. Via the comparison, we try to assess how much value is added by using label propagation in our setting. We then qualitatively analyze the synsets in the existing thesaurus and the synset suggestions from our manual studies. We deduce a set of challenges that need to be coped with in order to provide good extension suggestions. From that, we analyze why fastText embeddings have generally performed much better than word2vec embeddings.

### 6.1. Baseline: k-nearest-neighbors of Synset Vector

Our label propagation approaches operate on word embeddings and add additional layers on top of them: Graph construction and label propagation. For comparison, we defined a baseline approach that directly uses the word embeddings. It has the same goal: Extending existing training synsets by predicting the synsets labels for unlabeled words from the text corpus. It is inspired by the “Synset Embeddings” concept by Rothe and Schütze (2015), but we define synset embeddings in a simpler way.

For every training synset, we average the word embeddings of the training words contained in this synset. We call this average vector “*synset vector*” (synset embedding). We label the  $k$  closest words of a synset vector with the synset vector’s label. If a word is in the  $k$ -neighborhood of multiple synset vectors, the closest synset vector applies, i.e. the one with the highest cosine similarity.

As an example, in Figure 6.1, the training synset consists of “wordA” and “wordB”. They are averaged to a common synset vector. Its closest two word vectors, “wordX” and “wordY”, are labeled with the synset. “wordZ” is not among the two nearest neighbors of the synset vector and therefore not assigned to the synset, although it is very close to “wordB”.

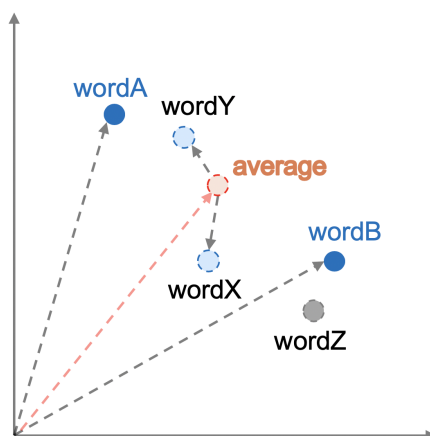


Figure 6.1.:  $k$ -nearest-neighbors of synset vector with  $k = 2$ . The training words of a synset determine an average vector. Words in its neighborhood get labeled with the respective label.

### 6.1.1. Quantitative and Qualitative Evaluation

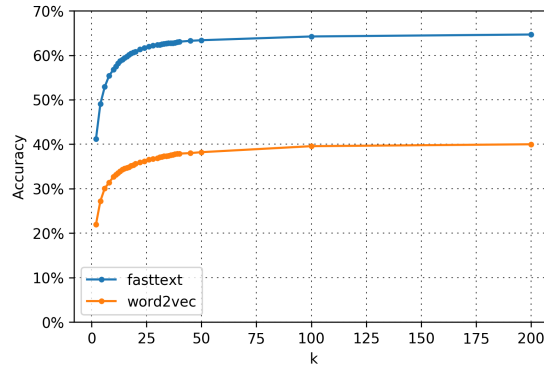
#### Quantitative Results

We compare the baseline with the fastText and word2vec word embeddings of our optimized pipeline configuration. Like in the parameter studies in Chapter 4, the evaluation was performed on three different training/test samplings of the thesaurus. The results over the three samplings were averaged.

In Figure 6.2, the accuracy results with varying parameter  $k$  are shown. For both embedding types, accuracy continuously increases with increasing  $k$ . This increase intuitively makes sense: All predictions of a run with  $k = n - 1$  stay the same for a run with  $k = n$ . Increasing  $k$  just means that each synset expands its range of words that get labeled with the respective synset. Words that already received a label for lower  $k$  stay in the same synset, because the synset's vector is still the closest one to the word. From around  $k = 25$ , the accuracy reaches around the same levels as the label propagation approaches. It does not change noticeably for higher  $k$  and converges at around 65%.

Table 6.1 shows a comparison between our optimized label propagation performances and the baseline for a high  $k$ ,  $k = 200$ . For fastText, the baseline reaches a slightly higher accuracy than our best label propagation configuration (65% against 62%). For word2vec, a slightly lower performance than label propagation is reached (40% against 41%). We also compare the top3 accuracy. For the baseline approach, we calculate this metric by storing the three closest synsets that have an individual word in their  $k$ -neighborhood, check if the correct test synset is included, and average over all test words. The top3 accuracies of baseline and label propagation are similar, although the label propagation approaches receive slightly higher values. Interestingly, the baseline



Figure 6.2.: Accuracy in a  $k$  parameter study for the synset vector baseline approach

reached this performance while, at the same time, not labeling some all words in the test set (the test set contained 2,887 words). For fastText, around 4% (124 words, averaged from three runs) of the test set words were not labeled. For word2vec, the number was even higher, with around 8% (238 words) test words not labeled. During label propagation, all test words were labeled.

	Accuracy	Top3 Accuracy		Accuracy	Top3 Accuracy
<i>fastText</i>	65%	77%	<i>fastText</i>	62%	79%
<i>word2vec</i>	40%	54%	<i>word2vec</i>	41%	56%

(a) Baseline ( $k = 200$ )

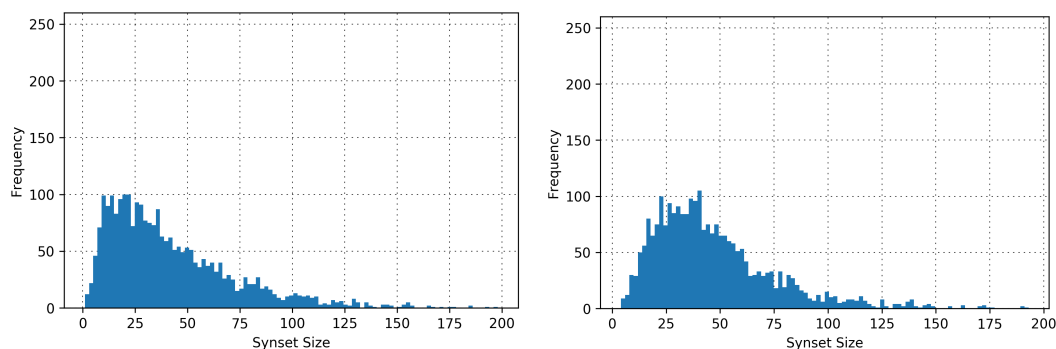
(b) Label propagation configurations

Table 6.1.: Accuracy and Top3 Accuracy for Baseline and Label Propagation compared

Figure 6.3 shows the synset size distributions.<sup>1</sup> Most interestingly, we noticed that in both cases, many words were not assigned a synset. For fastText, 81,511 words did not receive a label. For word2vec, the number is a bit lower with 68,060. From the word embeddings set, we know that we have around 190,000 words that can have a label and we have 2,552 different synsets. If all synsets expanded without interfering with each other, over 500,000 words could theoretically be labeled. As not even all 190,000 words received labels, we can see that many words are in the close neighborhood of multiple synsets at once, while many words are not in the neighborhood of any synset vector at all. In comparison, our label propagation approaches assigned a label to almost all words (except for 14 for word2vec). Therefore, the synset mean and median sizes are lower for the baseline than for the propagation approach. Comparing fastText and word2vec, the difference is similar to the label propagation distribution - fastText produces more smaller synsets than word2vec.

We see that our nearest-neighbor baseline resulted in around the same performance on

<sup>1</sup>Like in Chapter 4, the synset size distributions correspond to an individual thesaurus training/test split and are not averaged over multiple splits.



(a) fastText. Mean size is 41.74, median size is 34, max size 198. (b) word2vec. Mean size is 47.01, median size is 41, max size 193.

Figure 6.3.: Synset distributions of optimized configurations for fastText and word2vec embeddings (2552 different synsets)

the test set as our label propagation configurations. We verify this result by conducting manual evaluations on the baseline predictions as well.

### Qualitative Results

Like in Chapter 5, we manually evaluate now the quality of the baseline’s proposed predictions. We used the baseline with  $k = 30$  and trained with the full thesaurus.<sup>2</sup> For determining which predictions to show, we did not conduct a separate pre-study. Instead, we assumed that a high number of training words would correlate with good predictions as well. We selected 54 synsets with a high number of training words (greater or equal 80%-quantile) and showed the top 10 predictions, ordered by their similarity to the respective synset vector. We did not incorporate the of number of predicted words into our synset selection process. Instead, we just required a minimum number of 10 predicted words. We conducted an evaluation of the same prediction lists by two separate persons and averaged the results.

The results are shown together in Table 6.2 (rounded to one decimal), together with the propagation results. We see that baseline method generally performs better than the propagation, it receives more synonym ratings and less “not similar” ratings. The difference is not that big for fastText embeddings. For word2vec embeddings though, the baseline method receives much better ratings than the propagation approach. This is surprising, as we could not see that difference in performance in the quantitative evaluation. Rather, the label propagation approach for word2vec performed even slightly better. For word2vec, one of the two evaluations does not seem to be representative for its actual performance.

<sup>2</sup> $k = 30$  should be a sufficient value as we receive enough synsets with a size of 10 words.

Variant	Score (Share in %)		
	0	1	2
<i>Baseline: fastText</i>	2.7	61.5	35.8
<i>Propagation: fastText</i>	6.5	63.9	29.6
<i>Baseline: word2vec</i>	35.6	42.9	21,5
<i>Propagation: word2vec</i>	62.2	29.6	8.2

Table 6.2.: Main Study Results: Baseline vs. Propagation

## Summary

From our evaluations, we cannot see an advantage for the label propagation approach compared to the baseline method. Instead, the baseline even slightly over-performs label propagation with fastText in terms of accuracy, and generally performs better in the manual evaluation. The lower complexity of the baseline approach and the fact that it needs significantly less resources and computation time, are more advantages of this approach in comparison to our label propagation. As the order of performance is comparable, we want to learn how much the predictions actually differ from each other. It might be that label propagation actually returns almost the same predictions as our baseline and does not do anything significantly different.

### 6.1.2. Prediction Similarity Comparison

Here, we investigate the potential differences between the predictions of baseline and label propagation. We first compare the predictions on the test set for a specific training/test split. Then, we compare the predictions for selected synsets visually and try to learn if there are distinct patterns that differentiate label propagation and the baseline.

### Quantitative Comparison

We analyze how the predictions on the same test set differ from each other, depending on the method used. We compare four methods:

- Optimized Label Propagation Configuration, fastText (*lp\_ft*)
- Optimized Label Propagation Configuration, word2vec (*lp\_w2v*)
- Synset Vector Baseline with  $k = 200$ , fastText (*bl\_200\_ft*)
- Synset Vector Baseline with  $k = 200$ , wrord2vec (*bl\_200\_w2v*)

All methods were trained on the same training set and evaluated on the same test set (no averaging over multiple training/test splits took place). The size of the test set is 2887. *bl\_200\_ft* did not predict a label for 4% of the test data, *bl\_200\_w2v* for 9% of the

	lp_ft	lp_w2v	bl_200_ft	bl_200_w2v
lp_ft	100% / 61%	36% / 72%	65% / 73%	34% / 72%
lp_w2v		100% / 41%	35% / 75%	51% / 49%
bl_200_ft			100% / 65%	35% / 75%
bl_200_w2v				100% / 40%

Table 6.3.: Different methods compared for their prediction differences on test set. Left number: Share of equal test predictions, right number: Accuracy when combining predictions (one of the two methods matched the test).

test data. The results are shown in Table 6.3. Each table cell describes the intersection of the two methods given in row and column. The left number describes the share of equal test predictions. The right number shows the accuracy when combining the predictions of both methods, i.e. the share that at least one of the two methods matched a test. The table is symmetrical, therefore the values for the empty cells can be deduced by mirroring the existing values.

We note that a label propagation method and its baseline counterpart share a relatively high percentage of equal predictions. Combining the propagation and the baseline predictions produces a better accuracy than the methods individually. But it does not offer an significant advantage than when combining fastText and word2vec predictions of the same method. The combined accuracy still performs worse than the top3 accuracy metric of an individual method. Therefore, we conclude that label propagation does suggest different predictions than its baseline equivalent. But the different predictions do not seem to be valuable for an overall higher performance.

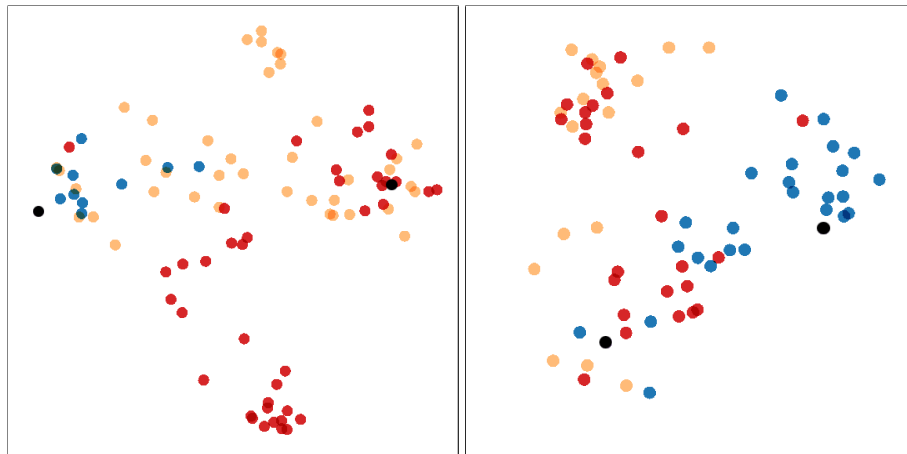
### Visual Comparison

In Table 6.3, we have seen that the propagation method, to a certain extent, identifies correct labellings that the baseline does not identify, and vice versa. We want to see if there are distinct visual pattern in how these two methods select their predictions. For fastText embeddings, we visualize two selected synsets and their closest 50 predictions within both methods. We use the Tensorflow Embedding Projector<sup>3</sup>, where the dimensionality is reduced from 400 to 2 via Principal Component Analysis (PCA). The visualizations are shown in Figure 6.4.

From our intuition, we expected the baseline to predominantly predict words in the direct neighborhood of the training words. We expected the propagation method’s predictions to include chain-like structures that are not explainable via direct neighborhood.

The visualizations in Figure 6.4 do no resemble these intuitions. We cannot identify a clear visual difference in the prediction behavior of the baseline and the propagation

<sup>3</sup><http://projector.tensorflow.org> visited on Nov. 3, 2018



(a) Synset predictions with “zinshöhe” and “zahlungshöhe” as training data  
 (b) Synset predictions with “milchverarbeitung” and “milchwirtschaft” as training data

Figure 6.4.: Two synsets and their predictions visualized. Black: Training words, Blue: Predicted by both propagation and baseline, Yellow: Propagation-only prediction, Red: Baseline-only prediction.

method. On the contrary, in the left plot, we see a group of points that is far away from both training words, but was suggested by the baseline, although there are many more points that are closer to the training words. We also see in the left plot that all common predictions are close to only one of the training words. In the right plot, this pattern is not as strong. However, also here we see that considerably more common predictions are close to one of the two training words. In general, in the right plot, we cannot see such a strong distinction into groups of “predicted by both” and “predicted by one specific method” like in the left visualization.

An explanation for that could be the dimensionality reduction. With PCA, only around 30% of the variance could be described. The remaining variance could just not be enough to show the expected behavior. We conclude that a comparison by visualization is difficult to use for finding differentiating patterns of propagation and baseline prediction behavior.

### 6.1.3. Application of Propagation to Baseline

We have seen that the synset vector baseline on its own results in good performance, both on a test set as well as via human rating. As an experiment, we apply label propagation on the baseline graph and evaluate whether we can increase performance. For that, we interpret the synset vector baseline as a similarity graph. In this model, synset vectors become graph nodes. We add edges from each synset vector to the  $k$  nearest

words, where the edge weight corresponds to the cosine similarity. Here, we use  $k = 200$ . As initial graph labeling, we label the synset nodes with the synset’s labels, while all word nodes become unlabeled. Then, we apply LabelSpreading with  $\alpha = 0.2$  and 15 iterations.

The results are shown in Table 6.4. By applying label propagation on it, we really can improve the performance of the baseline itself, although slightly. For *fastText*, accuracy could be increased by 1 percent point, top3 by 2 percent points. The increase is higher for *word2vec* embeddings, where 4 percent points for accuracy and 3 percent points for top3 accuracy could be gained.

	Accuracy	Top3 Accuracy		Accuracy	Top3 Accuracy
<i>fastText</i>	66%	79%	<i>fastText</i>	65%	77%
<i>word2vec</i>	44%	57%	<i>word2vec</i>	40%	54%

(a) Baseline with added Label Propagation

(b) Baseline ( $k = 200$ )

Table 6.4.: Accuracy and Top3 Accuracy for Baseline with added Label Propagation, compared to Baseline itself

## 6.2. Challenges around Thesaurus Extension

By analyzing the existing thesaurus and the synset suggestions we rated in our manual studies, we noted several aspects around thesaurus extension that make it challenging to find the right synset extensions. They are shown and categorized in Table 6.5. There are certain *semantic challenges*: Depending on the context, a word’s meaning can be different. If there are only few training words, an algorithm might suggest words that do not fit the anticipated word’s meaning. Another problem is connected to compound words. Often in such a compound word, one word is more defining than the other, so the less defining word should be varied to generate semantically similar suggestions. The German language that we analyze has many of such compound words. But in comparison to other languages, this is a rather rare phenomenon. So this problem might not be that relevant any more when trying to extend a thesaurus which is not German. Sometimes, the algorithm did not cope with this issue and suggested e.g. “us-börse” for a concept around “milchwirtschaft”. Furthermore, we recognized that it is very hard to decide if a broader term should be included in a synset with more specific terms, or even more difficult, if a more specific word like “einkommenssteuerrecht” should be added to a synset around “steuerrecht”.

Even more than semantic challenges, we noted several *syntactic challenges*. In the existing thesaurus, many entries of a synset were made up of inflections of the same words, words with the same word stem, different word splits or hyphenations or old spellings/misspellings of a word. This applies also to the other kinds of syntactically

Category	Type	Example
<i>Semantic Challenges</i>	Context-dependent word meaning	leiter (ladder vs. manager)
	Identification of defining word parts	milchwirtschaft (“milch” is more defining)
	Broader or more specific terms	steuerrecht, einkommenssteuerrecht
<i>Syntactic Challenges</i>	Inflected words	zeitungsträgern, zeitungsträger
	Same word stem	stornierung, stornieren
	Word splits	eigentümerehegatten, eigentümer ehegatten
	Hyphenation	zwölfmonatszeitraum, zwölfmonats-zeitraum
	Old spellings/Misspellings	fitneß-studios, fitness-studio
	Abbreviations	ustk, ust-kartei
	Numbers	12-monatsfrist, zwölfmonatsfrist

Table 6.5.: Challenges around Thesaurus Extension

similar entries - abbreviations and use of numbers. Our algorithms could often find more syntactically similar suggestions. While this is good for the algorithms’ performance, the question occurs if finding syntactic variations is the goal of thesaurus extension. Slight syntactic variations of words are not as interesting as synset suggestions. Instead of using machine learning techniques, suggestions like these could be automatically generated using more simple heuristics.

**Why fastText performs better than word2vec** The provided thesaurus contained many synset entries that are only syntactic variations of each other. Also, in the qualitative evaluation, we have given high ratings for these type of syntactic variations. These facts can explain why fastText embeddings have continuously received higher ratings than word2vec embeddings our evaluations. By examining the actual predictions, we can see that fastText places great weight on *syntactic similarity*. Words that are syntactically similar are very likely to get assigned to close vectors. word2vec more often offers suggestions not only based on syntactic similarity. With fastText embeddings, most suggestions are slight variations of words that are already in the synset. An example is shown in Table 6.6. The existing thesaurus that is used as a test set in the quantitative evaluation, as well as our qualitative evaluation, seem to reward the behavior of fastText’s embeddings.

<b>Existing Synset Words</b>	<b>fastText Propagation (Top 5)</b>	<b>word2vec Propagation (Top 5)</b>
<i>kst-bescheid</i>	körperschaftsteuer-bescheids	erstattungsjahre
<i>kst-bescheide</i>	kst-bescheiden	leistungsgebote
<i>körperschaftsteuer-bescheid</i>	körperschaftsteuer-bescheide	vek-bescheide
<i>körperschaftsteuerbescheid</i>	körperschaftsteuerbescheide	zuwendungsbestätigungsempfänger
	körperschaftsteuerbescheiden	umsatzsteuervorauszahlungsbescheide

Table 6.6.: Comparison between fastText and word2vec propagation results for a given synset



## 7. Conclusion and Future Work

**Implementation** Our first achievement was the introduction of a pipes and filters architecture that modeled the entire thesaurus extension process. It splits the problem into several, successive steps. For each step, hyper-parameters can be set, that affect the step's output. The pipeline is optimized for running many different hyper-parameter configurations and calculating performance metrics. Each step of the pipeline can be easily reimplemented or replaced, as long as input and output stay the same. Therefore, the pipeline can serve as a base for further research, e.g. using other graph construction approaches, or more sophisticated label propagation algorithms. Also, as the architecture is independent from the problem domain, it could be used to evaluate thesaurus extension in different contexts.

**Quantitative Evaluation** The crucial achievement of this chapter was the identification of optimal hyper-parameter configurations through parameter-studies. We tried to cover all important steps of our pipeline: pre-processing, embeddings generation, graph construction, and label propagation. Although there exist more label propagation and other graph-based machine learning approaches, we think it is sufficient to focus on the basic algorithms shown here (LabelPropagation and LabelSpreading). The overall success is depending on other properties.

While we feel as well quite comfortable with our work in the pre-processing phase and the embedding generation, especially in the field of graph construction further research would have been possible. Our results support this hypothesis: Pre-Processing and embedding generation benefit all steps (and alternative naive approaches which operate directly on the embeddings), not just label propagation. Graph construction is therefore the crucial step which determines the method's performance. Thus, future work should definitely investigate more approaches or variations instead of only considering standard similarity graphs. For example, Ravi and Diao (2015) propose a method to combine two different types of graphs into an *augmented graph*: A knowledge graph like from Freebase<sup>1</sup> and a graph that is constructed out of word embeddings. They claim that augmenting the Freebase graph with embeddings results in significant improvements in quality, when applying graph-based semi-supervised learning algorithms like label propagation. While Freebase was shut down, its data is still available for download. An

---

<sup>1</sup>Freebase was a free structured knowledge base for semantic data that was acquired by Google and shutdown in 2016. See <http://www.freebase.com>, visited on Nov. 3, 2018

alternative knowledge base would be Wikidata<sup>2</sup>.

**Qualitative Evaluation** Let us briefly recap why we think that a manual evaluation of the predictions is necessary and important: We wanted to make sure that a good performance on the thesaurus can be generalized to high performance on words in the entire text corpus. We have to keep in mind that the thesaurus was created manually. Therefore, it could be the case that only words with rather direct relations became part of the initial synsets. Thus, a quantitative analysis alone seems to be insufficient for us. And indeed, word2vec performs much worse in the qualitative evaluation than fastText, in a degree we had not expected from the quantitative evaluation. We could therefore support our claim from the previous chapter, that fastText predictions perform better than word2vec one's.

Although the design of the qualitative evaluation is debatable, we think it gives a good first impression on the validity of our results. We selected synsets via their number of training and predicted words and then sorted them according to their confidence. Instead, we could have included the confidence already in the synset selection process. By that, we could avoid selecting synsets where all predictions have a comparably low confidence. As a consequence, we could decrease the number of low human ratings and instead focus on promising suggestions.

During the entire process, we noted several aspects around thesaurus extension that make it challenging to identify the right synset suggestions. Although it uses much more resources, our complex label propagation algorithm was not able to overcome these challenges better than our rather simple synset vector approach.

### Final Remarks

The main idea of this thesis was to determine whether or not the theoretical idea to combine the concepts of word embeddings, graph construction and label propagation leads to good results for thesaurus extension. Our work and the subsequent, detailed analysis showed that we have to refute our hypothesis, that the effort is worth it.

What could be the reasons for that? We do not have a definite answer for that, but want to outline three possible reasons:

- German Language
- Context of Tax Law
- Global Consistency Attribute

---

<sup>2</sup><https://www.wikidata.org>, visited on Nov. 3, 2018

---

**German Language** Many semantic problems are language-specific. Especially, compound words do not exist in many other languages, which would make word distinction much easier. Each word would get taken into account separately and included within the context of other words. Therefore, it would be interesting to compare our results with a study that has a similar thesaurus and context, but in a different language. Another problem that could be maybe solved by that is the relatively low number of training data, although the data set is fairly large for a German one. In our thesaurus, after pre-processing, the majority of synsets had only two members. It is possible that e.g. in English, there could be much larger and qualitatively comparable thesauri. Better data in the training phase could have a significant influence on the quality of the results. An example for the availability of larger data sets is the recent publication of millions of U.S. court decisions by the “Caselaw Access Project”<sup>3</sup>, with the goal to make analysis on it possible. Such a large publication in German language is not likely because of the much lower population. Evaluating our research on an English data set would therefore be an interesting option.

**Context of Tax Law** Another potential reason could be the domain of tax law. Maybe, this domain is structurally not suitable for thesaurus extension using label propagation. An indication for this are the positive examples for a combination of word embeddings and label propagation we noted in the introduction, e.g. by Google for learning emotion associations. Therefore, it could be interesting to evaluate what happens if we extend a thesaurus from another domain. An example would be medicine. There, old patient’s records and typical courses of diseases are used to figure out new treatment plans. Thus, from a user’s perspective, the scenario is comparably to the one of law. But one could also choose a completely different context and analyze prose texts. It could be an advantage, as well as a disadvantage that prose texts, compared to technical texts, contain much more paraphrasing, and less information. An analysis could give interesting insights on word embeddings, and word embeddings benefit all subsequent steps, as we have discussed before.

**Global Consistency in Graph** However, we think that the most significant reason is the apparent absence of global consistency within our similarity graph. From Section 2.4, we know that the advantage of label propagation compared to nearest-neighbors algorithms is the fact that it takes the overall structure into account. It works for problems that satisfy the global consistency assumption, where points on the same structure are likely to have the same label. However, performance was about the same for label propagation and our nearest-neighbor baseline. Therefore, the global consistency assumption seems not to hold for the graphs we constructed during this thesis. It would be valuable to explore more graph approaches, potentially not constructed just from word embeddings, but from a combination of multiple knowledge bases.

---

<sup>3</sup><https://case.law>, visited on Nov. 6, 2018



## A. Proof: Euclidean instead of Cosine Distance for Word Embeddings

When dealing with word embeddings, vectors of similar words are close with respect to *cosine distance* (that is  $1 - \text{cosine similarity}$ ), not with respect to euclidean distance. To generate a k-nearest-neighbor graph out of the embeddings for later application of label propagation, we need to calculate these cosine distances. Unfortunately, implementing knn with cosine distance in scikit-learn resulted in memory errors. But: We can circumvent this problem and still use euclidean distances to get to the same knn result.

*We will show that cosine distance is directly related to the euclidean distance on normalized vectors (unit length). Therefore, normalizing the embeddings will return the same distance ordering for euclidean distance <sup>1</sup> as it would for cosine distances.*

We know about *Euclidean Distance*:

$$\text{euc}_{dist}(A, B) = \|A - B\| \tag{A.1}$$

$$= \sqrt{(A - B)^T(A - B)} \tag{A.2}$$

And about the *Cosine Distance*:

$$\text{cos}_{dist}(A, B) = 1 - \cos(A, B) \tag{A.3}$$

$$= 1 - \frac{A \cdot B}{\|A\| \|B\|} \tag{A.4}$$

$$= 1 - \frac{A^T B}{\|A\| \|B\|} \tag{A.5}$$

$$= 1 - A^T B \quad | \text{ As A and B have unit length} \tag{A.6}$$

---

<sup>1</sup>Same ordering, not the same distance values

From A.2 we get:

$$\text{euc}_{dist}(A, B) = \sqrt{\|A\|^2 - 2A^TB + \|B\|^2} \quad \left| \text{As } (A - B)^T(A - B) = \sum (a_i - b_i)^2 \quad (\text{A.7}) \right.$$

$$\left| = \sum a_i^2 - 2 \sum a_i b_i + \sum b_i^2 \quad (\text{A.8}) \right.$$

$$= \sqrt{2(1 - A^TB)} \quad \left| \text{As A and B have unit length} \quad (\text{A.9}) \right.$$

And we can insert the cosine distance from A.6:

$$\text{euc}_{dist}(A, B) = \sqrt{2 \cos_{dist}(A, B)} \quad (\text{A.10})$$

We see: If two vectors have a larger cosine distance than two others, they will, when normalized, also have a larger euclidean distance.

## B. Difference between two LabelPropagation Algorithms by Zhu et al.

Here, we discuss the difference for the LabelPropagation algorithm between the original paper (Zhu and Ghahramani 2002) and Zhu et al. (2005), Zhu’s doctoral thesis, where the algorithm was given in a modified version. In Zhu and Ghahramani (2002), a transition matrix field  $T_{ij}$  describes the probability of jumping from node  $j$  to  $i$ . This is calculated by normalizing the *columns*:

$$T_{ij} = P(j \rightarrow i) = \frac{w_{ij}}{\sum_{k=1}^n w_{kj}} \quad (\text{B.1})$$

In comparison to that, Zhu et al. (2005),  $T_{ij}$  corresponds to the probability of jumping from *node  $i$  to  $j$* , so normalizing the *rows*<sup>1</sup>:

$$T_{ij} = P(j \rightarrow i) = \frac{w_{ij}}{\sum_{k=1}^n w_{ik}} \quad (\text{B.2})$$

As the transition matrix from B.2 is automatically row-normalized, applying it iteratively to  $Y$  will result in all rows of  $Y$  having 1 as sum, too, so  $Y$  can be directly interpreted as a class probability matrix without explicit adjustment. In comparison to that, equation B.1 requires the explicit row-normalization of  $Y$  during each iteration to maintain the class probability interpretation. The repetitive row-normalization step can be bypassed by row-normalizing  $T$  before iterating.

The two different calculations of the transition matrices lead to a slightly different propagation. As Bodó and Csató (2015) observed, equation B.1 assigns greater weight to points having *fewer/distant* neighbors. Following on our example graph from Section 2.4.3, this becomes more clear. Figure B.1 shows the two transition matrices<sup>2</sup> and their respective  $Y^{(\infty)}$ . The algorithms converge to different class distributions for nodes 3 and 4 - for Zhu and Ghahramani (2002), class  $A$  is valued stronger. This is because node 1, where class  $A$  is “flowing from”, has just one neighboring node, compared to node

<sup>1</sup>Row-normalization of a quadratic matrix  $M$  can be achieved by  $D^{-1}M$ , where  $D$  is the (weighted) diagonal degree matrix, like we did it in algorithm 1.

<sup>2</sup>Because our affinity matrix  $W$  is symmetric, the non-row-normalized transition matrix from Zhu and Ghahramani (2002) is just the transposed transition matrix from Zhu et al. (2005).

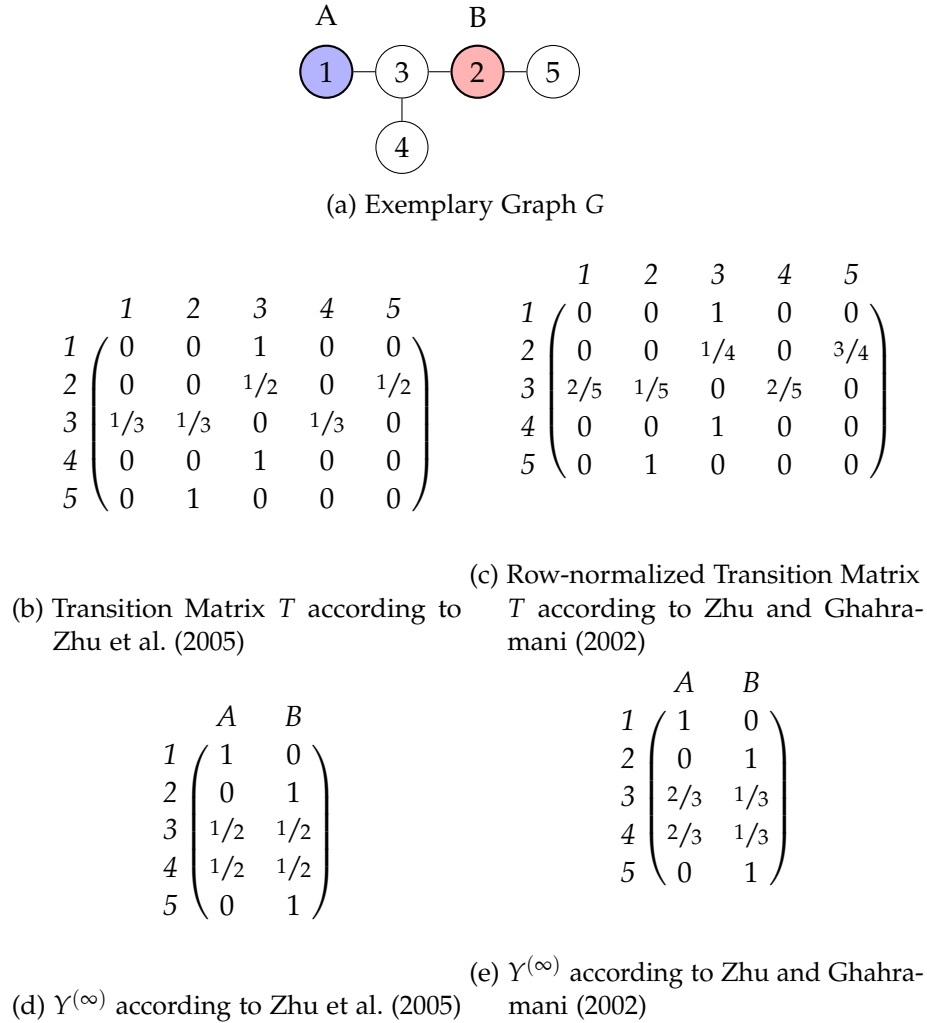


Figure B.1.: Differently calculated transition matrices and respective  $Y^{(\infty)}$  for exemplary graph G

2, root for class B, with two neighboring nodes. The label from node 1 therefore has a greater influence on the nodes 3 and 4 than the one from node 2.

A clear statement on which of these two algorithms performs better, is difficult. In our example, both resulting class probabilities could be intuitively justified. Bodó and Csató (2015) recommend the version by Zhu and Ghahramani (2002), as it results in a “natural, more balanced labeling”. On the other hand, most well-cited research seems to use the version by Zhu et al. (2005). Therefore, we chose to use the latter version as well.



# List of Figures

1.1. Demonstration of an automatic search query expansion . . . . .	3
1.2. The main activities around thesauri . . . . .	4
1.3. Sample word embeddings for a German tax law corpus visualized . . . . .	5
1.4. Issue with using the existing synset's nearest neighbors as candidates . . . . .	6
1.5. Intuition of Combining Word Embeddings with Label Propagation . . . . .	7
1.6. Our design science research process visualized . . . . .	8
2.1. Distribution of the different document types in the provided text corpus . . . . .	10
2.2. Histogram visualizing synset sizes in the untouched thesaurus . . . . .	10
2.3. word2vec CBOW and Skip-gram training model architectures . . . . .	13
2.4. Sample k-nearest-neighbor and $\epsilon$ -neighborhood graph construction . . . . .	15
2.5. Classification comparison between k-nearest-neighbors and LabelSpreading . . . . .	17
2.6. Intuition how label information flows out of labeled into unlabeled nodes . . . . .	17
2.7. Exemplary graph $G$ with five nodes . . . . .	18
2.8. LabelPropagation transition matrix for graph $G$ . . . . .	20
2.9. LabelSpreading transition matrix for graph $G$ . . . . .	22
3.1. Pipeline architecture that splits the problem into successive steps . . . . .	26
3.2. Demonstration of pre-processing on a simplified text from the corpus . . . . .	28
3.3. Pre-Processing Demonstration on single thesaurus synset . . . . .	32
3.4. Histogram on the distribution of synset sizes after thesaurus pre-processing . . . . .	34
4.1. Quantitative Evaluation as a Training and a Test phase . . . . .	37
4.2. Accuracy of word2vec, fastText and GloVe technologies compared . . . . .	39
4.3. Accuracy with varying dimensionality and iteration number compared. . . . .	40
4.4. Accuracy for knn-Graph and $\epsilon$ -graph compared . . . . .	40
4.5. Accuracy for Self-References and Edge Type Parameters Compared . . . . .	41
4.6. Accuracy with varying label propagation technology & iteration number . . . . .	42
4.7. Accuracy for LabelSpreading with varying $\alpha$ . . . . .	43
4.8. Accuracy with varying pre-processing settings . . . . .	43
4.9. Synset distributions of optimized configurations . . . . .	45
5.1. Screen shot of a manually evaluated synset . . . . .	47
5.2. Pre-Study Selection Procedure . . . . .	48
5.3. Correlation analysis between high rating and prediction characteristics . . . . .	49

6.1. Synset Vector baseline intuition visualized . . . . .	54
6.2. Accuracy in a $k$ parameter study for the synset vector baseline approach	55
6.3. Synset distributions of optimized configurations . . . . .	56
6.4. Two synsets and their predictions visualized in vector space . . . . .	59
B.1. Differently calculated transition matrices and $Y^{(\infty)}$ for graph $G$ . . . . .	70

# List of Tables

2.1. Corpus document size statistics . . . . .	9
2.2. Thesaurus synset size statistics . . . . .	10
3.1. Corpus Pre-Processing: Hyper-Parameters and possible values . . . . .	29
3.2. Effects of $\beta$ -handling and "Keep characters" options . . . . .	29
3.3. Embeddings Generation: Hyper-Parameters and possible values . . . . .	30
3.4. Graph Construction: Hyper-Parameters and possible values . . . . .	31
3.5. Thesaurus Pre-Processing: Hyper-Parameters and possible values . . . . .	33
3.6. Filtering out synsets and keys during thesaurus pre-processing . . . . .	33
3.7. Thesaurus Sampling: Hyper-Parameters and possible values . . . . .	34
3.8. Label Propagation: Hyper-Parameters and possible values . . . . .	36
4.1. Modifiable parameters per pipeline phase and their default values . . . . .	38
4.2. Accuracy and Top3 Accuracy of optimized configurations . . . . .	44
5.1. Main Study Results . . . . .	51
6.1. Accuracy and Top3 Accuracy for Baseline and Label Propagation compared . . . . .	55
6.2. Main Study Results: Baseline vs. Propagation . . . . .	57
6.3. Different methods compared for their prediction differences on test set . . . . .	58
6.4. Baseline+Label Propagation compared to Baseline . . . . .	60
6.5. Challenges around Thesaurus Extension . . . . .	61
6.6. Comparison between propagation results for given synset . . . . .	62



# Bibliography

- Baluja, S., R. Seth, D. Sivakumar, Y. Jing, J. Yagnik, S. Kumar, D. Ravichandran, and M. Aly (2008). "Video Suggestion and Discovery for Youtube: Taking Random Walks Through the View Graph." In: *Proceedings of the 17th International Conference on World Wide Web*. WWW '08. New York, NY, USA: ACM, pp. 895–904. ISBN: 978-1-60558-085-2. DOI: 10.1145/1367497.1367618 (cit. on p. 23).
- Baroni, M., G. Dinu, and G. Kruszewski (2014). "Don't Count, Predict! A Systematic Comparison of Context-Counting vs. Context-Predicting Semantic Vectors." In: *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Vol. 1, pp. 238–247 (cit. on p. 12).
- Bengio, Y., O. Delalleau, and N. Le Roux (Sept. 22, 2006). "Label Propagation and Quadratic Criterion." In: *Semi-Supervised Learning*. DOI: 10.7551/mitpress/9780262033589.003.0011 (cit. on pp. 6, 16, 18, 20, 21, 23, 36).
- Bengio, Y., R. Ducharme, P. Vincent, and C. Jauvin (2003). "A Neural Probabilistic Language Model." In: *Journal of machine learning research* 3 (Feb), pp. 1137–1155 (cit. on p. 12).
- Bodó, Z. and L. Csató (2015). "A Note on Label Propagation for Semi-Supervised Learning." In: *Acta Universitatis Sapientiae, Informatica* 7.1, pp. 18–30 (cit. on pp. 20, 69, 70).
- Bojanowski, P., E. Grave, A. Joulin, and T. Mikolov (2017). "Enriching Word Vectors with Subword Information." In: *Transactions of the Association for Computational Linguistics* 5, pp. 135–146. ISSN: 2307-387X (cit. on p. 11).
- Buchnik, E. and E. Cohen (Mar. 7, 2017). "Bootstrapped Graph Diffusions: Exposing the Power of Nonlinearity." In: arXiv: 1703.02618 [cs] (cit. on pp. 18, 20, 23, 36).
- Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal (1996). "A System of Patterns: Pattern-Oriented Software Architecture." In: (cit. on p. 25).
- Clark, S. (2015). "Vector Space Models of Lexical Meaning." In: *The Handbook of Contemporary Semantic Theory*. Wiley-Blackwell, pp. 493–522. ISBN: 978-1-118-88213-9. DOI: 10.1002/9781118882139.ch16 (cit. on p. 11).
- Conrad, J. G. and Q. Lu (Mar. 28, 2013). *Next Generation Legal Search – It's Already Here*. URL: <https://blog.law.cornell.edu/voxpath/2013/03/28/next-generation-legal-search-its-already-here/> (visited on Oct. 13, 2018) (cit. on p. 2).
- Deerwester, S., S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman (1990). "Indexing by Latent Semantic Analysis." In: *Journal of the American society for information science* 41.6, pp. 391–407 (cit. on p. 12).

- Dirschl, C. (Feb. 25, 2016). "Thesaurus Generation and Usage at Wolters Kluwer Deutschland GmbH." In: *Jusletter IT 25. Februar 2016* (cit. on pp. 3, 4).
- Grabmair, M., K. D. Ashley, R. Chen, P. Sureshkumar, C. Wang, E. Nyberg, and V. R. Walker (2015). "Introducing LUIMA: An Experiment in Legal Conceptual Retrieval of Vaccine Injury Decisions Using a UIMA Type System and Tools." In: *Proceedings of the 15th International Conference on Artificial Intelligence and Law. ICAIL '15*. New York, NY, USA: ACM, pp. 69–78. ISBN: 978-1-4503-3522-5. DOI: 10.1145/2746090.2746096 (cit. on p. 2).
- Grefenstette, G. (Jan. 1994). *Exploration in Automatic Thesaurus Discovery* (cit. on p. 14).
- Harris, Z. S. (1954). "Distributional Structure." In: *Word* 10.2-3, pp. 146–162 (cit. on pp. 4, 5, 11).
- Hevner, A. and S. Chatterjee (2010). "Design Science Research in Information Systems." In: *Design Research in Information Systems: Theory and Practice*. Boston, MA: Springer US, pp. 9–22. ISBN: 978-1-4419-5653-8. DOI: 10.1007/978-1-4419-5653-8\_2 (cit. on p. 8).
- Hinton, G. E., J. L. McClelland, and D. E. Rumelhart (1986). "Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1." In: ed. by D. E. Rumelhart, J. L. McClelland, and C. PDP Research Group. Cambridge, MA, USA: MIT Press, pp. 77–109. ISBN: 978-0-262-68053-0 (cit. on p. 11).
- Kannan, A., K. Kurach, S. Ravi, T. Kaufmann, A. Tomkins, B. Miklos, G. Corrado, L. Lukacs, M. Ganea, P. Young, and V. Ramavajjala (June 15, 2016). "Smart Reply: Automated Response Suggestion for Email." In: arXiv: 1606.04870 [cs] (cit. on p. 23).
- Keener, R. W. (2010). "Estimating Equations and Maximum Likelihood." In: *Theoretical Statistics: Topics for a Core Course*. Ed. by R. W. Keener. Springer Texts in Statistics. New York, NY: Springer New York, pp. 151–194. ISBN: 978-0-387-93839-4. DOI: 10.1007/978-0-387-93839-4\_9 (cit. on p. 50).
- Kiela, D., F. Hill, and S. Clark (2015). "Specializing Word Embeddings for Similarity or Relatedness." In: *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pp. 2044–2048 (cit. on p. 14).
- Landthaler, J., B. Walth, P. Holl, and F. Matthes (2016). "Extending Full Text Search for Legal Document Collections Using Word Embeddings." In: *JURIX*, pp. 73–82 (cit. on pp. 2, 12).
- Landthaler, J., B. Walth, D. Huth, D. Braun, C. Stocker, T. Geiger, and F. Matthes (June 16, 2017). "Extending Thesauri Using Word Embeddings and the Intersection Method." In: *Proceedings of 2nd Workshop on Automated Semantic Analysis of Information in Legal Texts. ASAIL '17*. London, UK (cit. on pp. 5, 9, 10, 14, 27).
- Lebret, R. and R. Collobert (2014). "Word Embeddings through Hellinger PCA." In: *Proceedings of the 14th Conference of the European Chapter of the Association for Computational Linguistics*, pp. 482–490 (cit. on p. 12).
- Meusel, R., M. Niepert, K. Eckert, and H. Stuckenschmidt (2010). "Thesaurus Extension Using Web Search Engines." In: *The Role of Digital Libraries in a Time of Global Change*.

- Ed. by G. Chowdhury, C. Koo, and J. Hunter. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 198–207. ISBN: 978-3-642-13654-2 (cit. on p. 14).
- Mikolov, T., K. Chen, G. Corrado, and J. Dean (Jan. 16, 2013a). “Efficient Estimation of Word Representations in Vector Space.” In: arXiv: 1301.3781 [cs] (cit. on pp. 5, 11–13).
- Mikolov, T., W.-t. Yih, and G. Zweig (2013b). “Linguistic Regularities in Continuous Space Word Representations.” In: *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 746–751 (cit. on p. 12).
- Miller, G. A., R. Beckwith, C. Fellbaum, D. Gross, and K. J. Miller (1990). “Introduction to WordNet: An on-Line Lexical Database.” In: *International journal of lexicography* 3.4, pp. 235–244 (cit. on p. 3).
- Nguyen, K. A., S. S. im Walde, and N. T. Vu (2016). “Integrating Distributional Lexical Contrast into Word Embeddings for Antonym-Synonym Distinction.” In: *arXiv preprint arXiv:1605.07766* (cit. on p. 14).
- OCDE, O.-O. (1996). “The Knowledge-Based Economy.” In: *Organisation for economic co operation and development, OeD, OECD* 2, pp. 1–46 (cit. on p. 1).
- Ono, M., M. Miwa, and Y. Sasaki (2015). “Word Embedding-Based Antonym Detection Using Thesauri and Distributional Information.” In: *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 984–989 (cit. on p. 14).
- Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay (2011). “Scikit-Learn: Machine Learning in Python.” In: *Journal of Machine Learning Research* 12, pp. 2825–2830 (cit. on pp. 23, 31).
- Pennington, J., R. Socher, and C. D. Manning (2014). “GloVe: Global Vectors for Word Representation.” In: *Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1532–1543 (cit. on pp. 11–13, 30).
- Ravi, S. and Q. Diao (Dec. 6, 2015). “Large Scale Distributed Semi-Supervised Learning Using Streaming Approximation.” In: arXiv: 1512.01752 [cs] (cit. on pp. 4, 23, 63).
- Řehůřek, R. and P. Sojka (May 2010). “Software Framework for Topic Modelling with Large Corpora.” In: *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. Valletta, Malta: ELRA, pp. 45–50 (cit. on p. 29).
- Rothe, S. and H. Schütze (2015). “AutoExtend: Extending Word Embeddings to Embeddings for Synsets and Lexemes.” In: *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Vol. 1, pp. 1793–1803 (cit. on pp. 14, 53).
- Rychlý, P. and A. Kilgarriff (2007). “An Efficient Algorithm for Building a Distributional Thesaurus (and Other Sketch Engine Developments).” In: *Proceedings of the 45th Annual Meeting of the ACL on Interactive Poster and Demonstration Sessions*. Association for Computational Linguistics, pp. 41–44 (cit. on p. 14).

- Salton, G. (1989). *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 978-0-201-12227-5 (cit. on p. 14).
- Schütze, H., C. D. Manning, and P. Raghavan (2008). *Introduction to Information Retrieval*. Vol. 39. Cambridge University Press (cit. on pp. 1, 2).
- Sparck Jones, K. (1964). *Synonymy and Semantic Classification*. Cambridge: Eng., Cambridge Language Research Unit (cit. on pp. 4, 13).
- “Spearman Rank Correlation Coefficient” (2008). In: *The Concise Encyclopedia of Statistics*. New York, NY: Springer New York, pp. 502–505. ISBN: 978-0-387-32833-1. DOI: 10.1007/978-0-387-32833-1\_379 (cit. on p. 50).
- Tai, Y.-J. and H.-Y. Kao (2013). “Automatic Domain-Specific Sentiment Lexicon Generation with Label Propagation.” In: *Proceedings of International Conference on Information Integration and Web-Based Applications & Services*. IIWAS '13. New York, NY, USA: ACM, 53:53–53:62. ISBN: 978-1-4503-2113-6. DOI: 10.1145/2539150.2539190 (cit. on p. 23).
- Takenobu, T., F. Atsushi, S. Naoyuki, and T. Hozumi (1997). “Extending a Thesaurus by Classifying Words.” In: *Automatic Information Extraction and Building of Lexical Semantic Resources for NLP Applications* (cit. on p. 14).
- Tamsin Maxwell, K. (Sept. 17, 2009). *Pushing the Envelope: Innovation in Legal Search*. URL: <https://blog.law.cornell.edu/voxpath/2009/09/17/pushing-the-envelope-innovation-in-legal-search/> (visited on Oct. 13, 2018) (cit. on p. 2).
- Ugander, J. and L. Backstrom (2013). “Balanced Label Propagation for Partitioning Massive Graphs.” In: *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*. WSDM '13. New York, NY, USA: ACM, pp. 507–516. ISBN: 978-1-4503-1869-3. DOI: 10.1145/2433396.2433461 (cit. on p. 24).
- Uramoto, N. (1996). “Positioning Unknown Words in a Thesaurus by Using Information Extracted from a Corpus.” In: *Proceedings of the 16th Conference on Computational Linguistics - Volume 2*. COLING '96. Stroudsburg, PA, USA: Association for Computational Linguistics, pp. 956–961. DOI: 10.3115/993268.993333 (cit. on p. 14).
- Von Luxburg, U. (Dec. 2007). “A Tutorial on Spectral Clustering.” In: *Statistics and Computing* 17.4, pp. 395–416. ISSN: 1573-1375. DOI: 10.1007/s11222-007-9033-z (cit. on pp. 15, 23).
- Wendt, J. B., M. Bendersky, L. Garcia-Pueyo, V. Josifovski, B. Miklos, I. Krka, A. Saikia, J. Yang, M.-A. Cartright, and S. Ravi (2016). “Hierarchical Label Propagation and Discovery for Machine Generated Email.” In: *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining*. WSDM '16. New York, NY, USA: ACM, pp. 317–326. ISBN: 978-1-4503-3716-8. DOI: 10.1145/2835776.2835780 (cit. on p. 23).
- Zhou, D., O. Bousquet, T. N. Lal, J. Weston, and B. Schölkopf (2004). “Learning with Local and Global Consistency.” In: *Advances in Neural Information Processing Systems*, pp. 321–328 (cit. on pp. 16–18, 21, 23, 35).
- Zhu, X. and Z. Ghahramani (2002). “Learning from Labeled and Unlabeled Data with Label Propagation.” In: (cit. on pp. 16, 17, 19–21, 69, 70).



Zhu, X., J. Lafferty, and R. Rosenfeld (2005). "Semi-Supervised Learning with Graphs." PhD Thesis. Carnegie Mellon University, language technologies institute, school of computer science (cit. on pp. 14, 17, 20, 21, 69, 70).