Word embedding models as graphs

Conversion and evaluation

Erik Winge

Thesis submitted for the degree of
Master in Informatics: Programming and networks
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Autumn 2018

Word embedding models as graphs

Conversion and evaluation

Erik Winge

Word embedding models as graphs

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

In this thesis, we make and evaluate procedures for converting between different lexical semantic representations. We implement three different methods for converting from vector space models to graph models: the threshold method, the kNN method and the variable-*k* method. We also implement a single procedure for converting from graph models to word embedding models. Further, we do comprehensive evaluation of our conversion procedures and their results. We perform extensive intrinsic evaluation using several gold standard data sets. We also do extrinsic evaluation of our converted graph models. We show that our graph models perform well at two real-world tasks: word sense induction/disambiguation and hypernym discovery.

The use of word embeddings has become widespread, in large part because of the increasing use of deep learning methods. However, training high quality word embeddings is time-consuming and requires large corpora. Therefore, an increasing number of pre-trained word embedding models have been made available, for instance by the Nordic language processing laboratory. These pre-trained word embeddings are useful in many natural language processing tasks. However, there are also many graph-based algorithms. We show that we can utilize high-quality word embeddings in graph-based algorithms by converting word embedding models to graphs.

Contents

List of Figures

List of Tables

Preface

I wish to thank my main supervisor Andrei Kutuzov, and co-supervisor Erik Velldal. Without their steady guidance, this thesis would not be the same. I would also like to thank the faculty and staff of the Department of Informatics, who have contributed to making the master's programme an enjoyable experience.

Chapter 1

Introduction

In this thesis, we investigate various ways to convert between different lexical semantic representations. Semantics is the systematic study of the meaning of language, both of individual words and sentences. We focus on word meaning in this thesis. Word meaning is also referred to as lexical semantics. The meaning of words has traditionally been presented in printed dictionaries as short definition texts, sometimes along with synonyms. However, textual definitions are of limited use in computer applications, such as natural language processing applications. Therefore, many forms of computer readable lexical semantic representations have been created. We focus on vector space models and graph-based models. These are among the most widespread contemporary models.

Both vector space models and graph models have long traditions in language technology. Osgood, Suci and Tannenbaum (1957) suggested using a Euclidean vector space to represent word meaning. Vector space semantic models represent words as vectors in a high-dimensional space. The axes of the space are the features we use to represent a word. The value for each element in a word vector is commonly a floating-point number, which indicates the frequency or magnitude of this feature for this word. Graph semantic models on the other hand, represent words as nodes in a graph. Different kinds of relationships between words are represented as edges between the nodes. Graph models are used in many algorithms. Semantic networks, like WordNet(Miller 1995), are examples of graph models that are widely used.

Vector and graph semantic models have different and somewhat complementary strengths and weaknesses. Graph models can contain a wider range of information than standard vector space models. For example, WordNet has a hierarchy of hypernym/hyponym[1] relationships, as well as many other relationships between words, which we discuss in section 2.3.2. In this way, WordNet explicitly models more relations than vector space models.

Lexical semantic models can be divided in two categories by how they are constructed, or their data source. These categories are distributional

[1]The hypernym/hyponym relation is the relation between a more general term, such as "animal", and a more specific, subordinate term, such as "cat".

semantic models and models that are constructed more or less manually. Manually constructed models can be built entirely by hand, like WordNet. Alternatively, they can be made with some degree of automation, but based on handcrafted features or annotations, for instance vector space models using handmade semantic features.

Distributional semantics (Harris 1954; Firth 1957) is the study of word meaning using the statistical distribution of words and their co-occurrences in a large corpus, that is a large collection of text. Distributional semantics uses unsupervised techniques, and can work with raw text with minimal and automatic preprocessing. No manual processing or annotation is required. In distributional semantics, the meaning of a word is somehow represented by its context. We can use various kinds of models and definitions of context. Different distributional semantic models represent the meaning of words in different ways. They can also represent different aspects of the meaning of a word, and different relationships between words. Such relationships can for instance be the hypernym/hyponym relationship. Distributional semantics, vector space models and graph models are discussed in further detail in chapter 2.

Word embeddings (Bengio et al. 2003; Mikolov, Chen et al. 2013), which are a kind of vector space models, have received much attention in the last few years. This is partly due to their use in artificial neural networks and deep learning. Word embedding models are vector space models with a lower number of dimensions, commonly in the order of hundreds. Word embeddings are also dense, vector elements are rarely zero. This low dimensionality makes them more efficient to use, which is beneficial in general, and is particularly important for neural networks.

An increasing number of pre-trained word embedding models have been made available on the web over the last years. It would be beneficial if we could convert these models to graph models, for use with graph-based algorithms. In this thesis, we investigate if this conversion can be done without significant loss of data. However, there are many ways of converting vector models to graph models. For instance, for large models it is expensive to produce fully connected graphs. Therefore, we evaluate different conversion methods and their hyperparameters.

We also explore converting the other way, from graph models to word embedding models. This is useful for making embeddings based on existing graph models. Such a graph model might for instance be a semantic network like WordNet, or a graph distributional semantic model.

To measure the effectiveness of our conversion procedures, we perform comprehensive evaluation. We do both intrinsic and extrinsic evaluation of our conversion results. We use both SimLex-999 (Hill, Reichart and Korhonen 2015) and other gold standard data sets in a thorough intrinsic evaluation. Additionally, we perform extrinsic evaluation in two different application settings. First, we evaluate our converted graph models in a word sense induction and word sense disambiguation task from SemEval-2013. Finally, we evaluate the converted graph models in an exploratory hypernym discovery task.

We show that both these conversions can be done without significant

information loss. We also demonstrate the usefulness of the resulting models in two application settings. Further, we discuss the pros and cons of the different conversion methods and hyperparameters.

We have made software to do the conversion and evaluation. Our software was implemented in Python, using standard open source libraries. All the project software is freely available on GitHub.[2]

The rest of this thesis is structured as follows: In chapter 2 we discuss some background material on distributional semantic models. We first introduce some terms and concepts from natural language processing and distributional semantics. Then we discuss vector and graph distributional semantic models in some detail.

In chapter 3 we introduce procedures for converting between vector and graph lexical semantic representations. We consider and implement different approaches for converting word embedding models to graph models. We also examine and implement conversion the other way, from graph models to word embedding models.

Chapter 4 contains a thorough intrinsic evaluation of the various conversion procedures that we have implemented. We use SimLex-999 and other word similarity based gold standard data sets to evaluate the quality of our converted models. We also evaluate the time and space use of our conversion procedures.

In chapter 5 we do an extrinsic evaluation in an application setting. The evaluation is done by means of a word sense induction and word sense disambiguation task. We implement word sense induction and disambiguation using our converted model. The result of the word sense disambiguation is then used to evaluate the quality of the conversion.

In chapter 6 we do further extrinsic evaluation, in another application setting. We explore whether our converted graphs can be used in hypernym discovery. To this end, we use subgraphs containing hypernyms and hyponyms from WordNet, and attempt to identify the hypernyms. This hypernym identification is done using node centrality information obtained from the graph.

In summary, in this thesis we make a bridge between two different lexical semantic representations. We convert vector space models to graph models, and vice versa. We evaluate the performance of our conversion procedures and the resulting models, by both intrinsic and extrinsic evaluation. Finally, we demonstrate that our converted graph models are useful in real-world tasks such as word sense induction and word sense disambiguation.

[2] https://github.com/ewinge/converter

Chapter 2

Distributional semantic models

Words are arbitrary labels for things and ideas in the real world. Since children can learn the meaning of words by being exposed to language, it should be possible for computers to do the same. *Distributional semantic models* are attempts at this. In this chapter, we investigate the current state of distributional semantic models. The main contenders are *vector space models* and *graph distributional semantic models*. We discuss the different variants of these models, and their strengths and weaknesses. Hopefully, this give some insight into a field that has seen rapid development in the last few years.

In the next section, we give some background on distributional semantics. In section 2.2, we discuss vector models, and in section 2.3 we discuss graph models. In section 2.4 we summarize this chapter. In the next chapter, we look at how these different architectures can be combined to get the best of both worlds.

2.1 Distributional semantics

The idea that the meaning of a word can be defined by or learned from the contexts it appears in goes back at least to Wittgenstein (Wittgenstein 1953). While he wrote mainly of the physical context, it has later come to mean a textual context. *Distributional semantics* is based on the *distributional hypothesis (DH)*, which is that words that are used in similar contexts, tend to have similar meanings (Harris 1954; Firth 1957). Firth was among the first to recognize this, and wrote the well-known quote: 'You shall know a word by the company it keeps!' (Firth 1957, p. 11). However, Firth's interest in the subject goes further back. In 1935 he seems to consider the contextual view of semantics to be an established principle, when he writes 'Secondly, the complete meaning of a word is always contextual, and no study of meaning apart from a complete context can be taken seriously.' (Firth 1935, p. 37).

Distributional semantics is a type of *statistical semantics*. The distributional hypothesis is the foundation of *distributional semantic mod-*

els (DSMs) used in computational linguistics. The two main approaches are *vector space models (VSMs)* and *graph distributional semantic models (GDSMs)*. They are both constructed with data-driven methods. Vector space models represent words as vectors in a high-dimensional space, where the dimensions are some kind of semantic features. Graph-based models represent words as nodes that are connected (related) by weighted edges. Both kinds of model are commonly generated by processing large amounts of text from a *corpus*. Normally, a larger and more varied corpus will yield a more accurate and representative model.

2.1.1 Tokens, types, full-forms and lexemes

When processing natural language, the text is first split or *tokenized* into separate pieces called *tokens*. A token is an atomic unit of text like a word, number or punctuation, usually separated by whitespace. However, some tokens, like punctuation, can follow other tokens immediately, without intervening whitespace. Contractions are often split into their separate components. For example, the text "I haven't seen her." contains the following six tokens: ["I", "have", "n't", "seen", "her", "."].

Tokens are instances of *types*, which are unique combinations of characters. The text "He had never had a car" contains six tokens, but only five types, because the type "had" is repeated. Tokens and types can be recorded in two forms, either *full-forms* or *lemmas*. A full-form is the inflected surface form as it occurs in the text. A lemma or *lexeme* is the uninflected dictionary form of a word. Because of ambiguity like homonymy, reducing a type to its corresponding lemma is not straightforward. The units used in distributional models are often full-forms, but can also be lexemes.

2.1.2 Corpus processing

A distributional semantic model reflects the corpus it is built on. When making an all-round model, it is important to gather a large corpus covering different genres and a wide range of topic matter. The corpus is generally processed sequentially. Before the corpus text is processed statistically, it undergoes linguistic preprocessing. The first step is *tokenizing*, splitting the text into *tokens* corresponding to words and other text units like numbers and punctuation. For most distributional semantic applications, punctuation is skipped. The tokenizer might also skip *stop words*, which are high-frequency function words like "an" or "it". Because they occur in nearly every context, stop words are of little use in discriminating contexts.

Next, each token might be normalized, for example by lowercaseing, *stemming* or *lemmatisation*. A *lemma* is the uninflected dictionary form of a word, while a *word stem* is a morphological unit where some or all suffixes have been removed. For example, "friendships" is lemmatized to "friendship", while a stemmer may yield either "friend" or "friendship" depending on the degree of stemming used. Lemmatisation requires a dictionary and morphological analysis. It does not handle words that are

not in the dictionary. Stemming is cruder but more efficient, mainly just chopping off the ends of words. It uses a set of rules detailing which suffixes should be removed. Stemming does not always result in proper words, for example "angrily" could yield "angri". Nor does it handle irregular inflection, like "bad"/"worst". Identical tokens identify a *term*, and normalizing maps several different tokens or words to the same term. The amount of normalizing to use depends on the application.

An optional, third step is annotation of the tokens. This adds information that is not explicitly part of the text, like word class tags or grammatical information. This can for example help differentiate between the verb-sense and noun-sense of ambiguous words like "cook".

2.1.3 Contexts

The foundation of all distributional semantic models is a set of observations from a corpus of text. These observations are made by processing the tokens in the corpus sequentially. Each observation consists of a *target term*, corresponding to the current token, and its *context*, consisting of one or more *context features*. How these observations are stored and processed are the main differences between different DSMs.

Various context types are used, for example the context can be the sentence or paragraph the token occurs in. *Bag of words (BoW)* contexts are unordered lists of N words surrounding the target word. *Context windows* are lists of $\pm n$ neighboring words, possibly weighted by distance. *Grammatical contexts* use grammatical relationships between the words, for example: "object-of(give)". This requires using a parser for deeper linguistic analysis. The different context types can also be used in combination, for instance a context window plus grammatical dependencies.

2.1.4 Measuring similarity

Measuring *word similarity* is an important application of distributional semantic models. The similarity function $sim(t_a, t_b)$ calculates the similarity of two terms t_a and t_b. The implementation of this function depends on the structure of the model. For graph models, the similarity is often the number of shared features. However, it can also be calculated by various graph metrics, such as the length of the shortest path between nodes. For vector space models the function measures the distance between points in the space. Additionally, there are *probability distribution models*, which use probability distributions for measuring similarity. All these models build on the distributional hypothesis. They all have three layers of abstraction, with the bottom two layers in common. These are "connected" by the distributional hypothesis:

$$\text{Meaning} \overset{DH}{\longleftrightarrow} \text{Contextual similarity} \longleftrightarrow \begin{cases} \text{Graph distance} \\ \text{Geometric distance} \\ \text{Prob. distribution divergence} \end{cases}$$

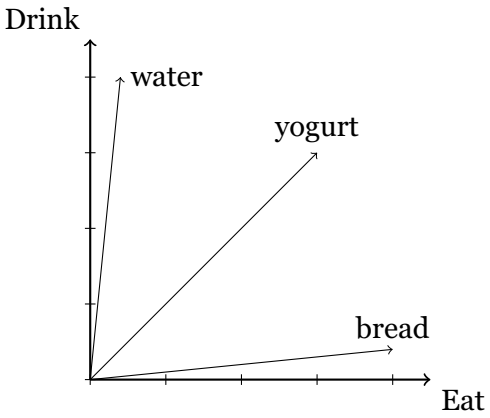


Figure 2.1: 2-D VSM with dimensions “Eat” and “Drink”

The top layer is the vectors, graph or probability distributions which can be used in applications.

Different ranges are used for the similarity function. Vector space models often use the range $[0, 1]$, where 0 means no similarity and 1 represents identical meaning. We can also use an unbounded similarity measure, with the range $[0, +\infty)$. Although the three models use different similarity functions, it is desirable that they give more or less the same relative ranking of similarity between word pairs. The ideal is that a similarity function should approximate the “true” semantic similarity of the terms.

2.2 Vector models

As the name implies, vector space models represent the meaning of words as vectors in a high-dimensional space. Each dimension represents a context feature. Figure 2.1 is a small example of a two-dimensional vector space model. It has the dimensions “eat” and “drink”, and shows that the term “water” co-occurs frequently with “drink”, while “bread” co-occurs with “eat”. While real VSMs have much larger numbers of vectors and dimensions, the general principles are the same.

2.2.1 Vector representations

Vector space models use a matrix that records the words and their contexts. Usually, the rows correspond to words (terms) in the vocabulary, and the columns correspond to contexts that the words occur in. The matrices are built by counting occurrences of words in each context. The main difference that separates VSMs, is their definition of the context. The two most common variations are *term–document* and *word–word* matrices, while *pair–pattern* matrices are a newer development (Turney and Pantel 2010).

Term–document matrices use entire documents as the context. The

rows are term-vectors which define terms by the documents they occur in. The columns are document-vectors which represent documents by the term used in them. Term–document matrices are common in information retrieval.

Word–word matrices register co-occurrences of words within shorter contexts. The type and size of the context varies, and can be for instance a window of ±100 words, or just a single sentence. Like in term–document matrices, the rows are also word-vectors. They represent the meanings of the words by the frequencies of other words they co-occur with. These vectors are somewhat similar to handmade semantic feature vectors. For example, a handmade vector for rooster could be (+bird, +male, -flighted, -young,...). The corresponding VSM vector could be (hen:231, egg:42, corn:78, coop:96,...). However, word vectors have much higher dimensionality than feature vectors that can reasonably be made by hand. A word–word matrix is of size $V \times C$, where V is the size of the vocabulary and C is the number of context features. For smaller models, the entire vocabulary can be used, so that $V = C$. But for models with a large vocabulary, the number of context words can be limited, for example to the 300 000 most frequent words. C is the number of dimensions in the VSM.

2.2.2 Weighting

Some context words are more informative or discriminating than others. For example, the word "sell" occur in the context of many things, like "car", "stock" and "book". Words like "drive" and "wheel" provide more information specific to "car". Thus it makes sense to give them more weight. A common method of weighing the terms in the matrix, is *Positive Pointwice Mutual Information (PPMI)* (Church and Hanks 1989; Turney and Pantel 2010). This is a measure of how often two words occur together, compared to the statistical expectation. However, PMI is biased towards infrequent words. One way to mitigate this, is to use Laplace smoothing to add a small constant to all counts in the matrix.

An alternative to PPMI is the *t-test* statistic, which comes from hypothesis testing. The test examines the mean and variance of samples, and tells if we can reject the null hypothesis that occurrences of two words are independent (C. D. Manning 1999).

For word–document models, *tf-idf (term frequency ×inverse document frequency)* weighting is widely used. The term frequency is multiplied by the idf, which is a measure of how common the term is in the corpus as a whole. $\text{idf}_t = \log(N/n_t)$, where N is the corpus size, and n_t is the number of documents containing the term t. This gives more weight to terms that occur in few documents, because they are probably more discriminative. Another weighting method used in information retrieval, is length normalization. This is necessary when the search favors longer documents.

2.2.3 Measuring similarity

Measuring *word similarity* is one of the main uses of word vectors. Each word vector represents a point in N-dimensional space. The closer two vectors point, the more similar are the contexts of the words they represent, and hence the words themselves, by the distributional hypothesis. The relative frequency of the words affect the length of the word vectors. In order to be able to compare frequent words, with longer vectors, to infrequent words, it is helpful to normalize all the vectors to unit length. Unit vectors all point to the unit hypersphere.

There are several different ways of calculating vector similarity. The *cosine* of the angle between vectors and the *Euclidean distance* between their endpoints are widely used. The cosine of the angle increases from -1 for vectors pointing in opposite directions, through 0 for perpendicular vectors to 1 for vectors pointing in the same direction. For unit vectors, the cosine is simply the dot product, $\vec{u} \cdot \vec{v}$. For example, in a small test corpus, the cosine similarity between the two words "pepper" and "salt" was 0.573, and the similarity between "pepper" and "California" was 0.032. While the absolute value of the numbers alone do not tell much, their relative difference matches the expectation. The Euclidean distance is the length of the vector difference $|\vec{u} - \vec{v}|$. With unit vectors, these two measures give the same ranking of the vectors.

2.2.4 Word embeddings

Vectors and matrices can be *sparse* or *dense*. Sparse vectors contain a high proportion of zeros. Dense word vectors are also known as *word embeddings*, from the mathematical concept of embedding, where one structure is mapped into another. While sparse vectors can have from thousands to millions of dimensions, word embeddings usually have only hundreds. The dimensions of sparse models have meaning, because they correspond directly to semantic features. However, the individual dimensions of word embeddings have no meaningful interpretation. The meaning of the vector is *distributed* among the dimensions. There is a many-to-many relationship between the dimensions and the semantic features. A feature can influence multiple dimensions, and each dimension represents many features. Word embeddings can either be produced by reducing the dimensionality of sparse vectors, or made directly.

2.2.5 Dimensionality reduction

After processing the text corpus, the resulting *raw* word-vectors have *high dimensionality*, often in the order of magnitude of 100 000. Because most words do not co-occur, most of the values will be zero, so the vectors are sparse. Thus the model will be more efficient if the number of dimensions can be reduced.

Traditionally, dimensionality reduction has been done by mathematical transformation of the co-occurrence matrix. *Singular Value Decomposi-*

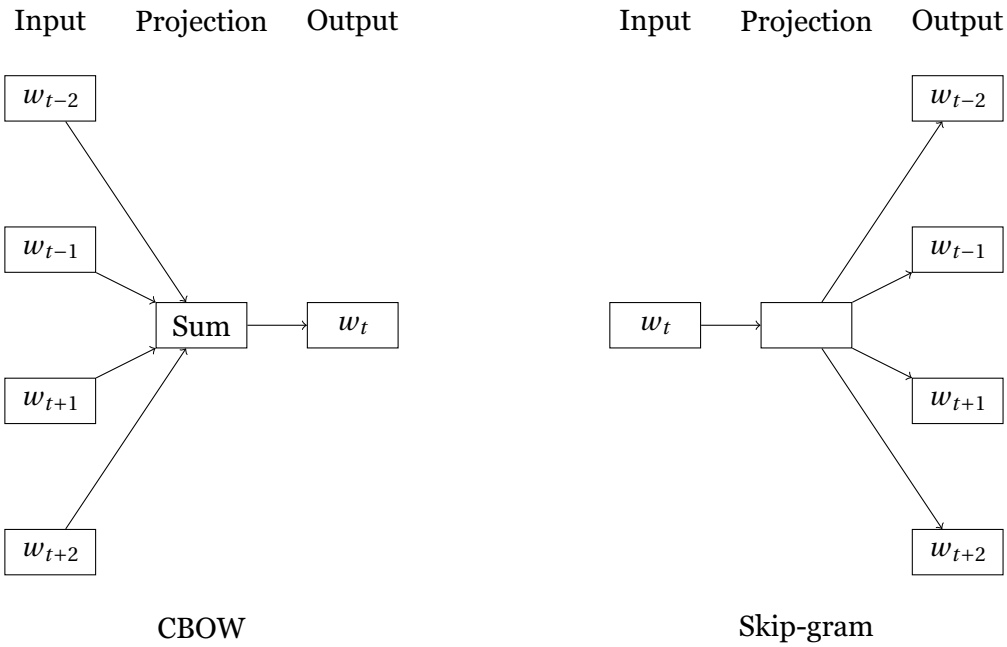


Figure 2.2: The two different word2vec architectures

tion (SVD) is a method for factorizing matrices. It is used in *Latent Semantic Analysis (LSA)* (Deerwester et al. 1990). LSA uses SVD to factorize the co-occurrence matrix, and then keeps only the k most significant dimensions, measured by variance. This is called *truncated* SVD.

There are numerous other methods of dimensionality reduction, including *Random indexing* (Kanerva, Kristofersson and Holst 2000), *NMF* (Lee and Seung 1999) and *t-SNE* (van der Maaten and Hinton 2008). Brown vectors are a special case, that do not use a co-occurrence matrix. *Brown clustering* (Brown et al. 1992) uses *class-based* language models to make a hierarchical clustering of the vocabulary. The paths in the resulting tree can be used as binary vectors to represent the words at the leaf nodes, similar to Huffman trees. One can also use a path to an internal node to represent the subtree/cluster rooted there, which gives dimensionality reduction.

2.2.6 Prediction based models

A new approach is to make word embeddings directly, without an intermediate sparse representation. Bengio et al. (2003) was among the first to use an artificial neural network for creating word embeddings. The concept was developed further in the word2vec software (Mikolov, Chen et al. 2013; Mikolov, Sutskever et al. 2013). In word2vec the neural network learns embeddings by iteratively improving an initially random vector, making it similar to the embeddings of context words. Word2vec comes in two varieties, the *continuous bag-of-word (CBOW)* and *skip-gram* models. Both models have input, projection and output layers. They differ in how they relate a word to its context, as fig. 2.2 shows.

Depending on the architecture, the word2vec neural network attempts to predict one or more output words from one or more input words. The word2vec objective is to maximize the conditional probability of an output word given an input word. In a naïve implementation, this conditional probability of output w_o given input w_i is calculated by the following softmax function:

$$p(w_o|w_i) = \frac{exp({v'_{w_o}}^{\top} v_{w_i})}{\sum_{w \in W} exp({v'_w}^{\top} v_{w_i})} \tag{2.1}$$

Here, v_w and v'_w are the input and output vectors or weights of the word w. The denominator consists of a sum over all terms in the vocabulary. This is very costly to compute. Therefore, word2vec is implemented with hierarchical softmax, or other optimizations. Hierarchical softmax replaces the output layer of the network with a tree structure. This reduces the computational cost, as we discuss further below.

Word2vec uses gradient descent to optimize the weights, with the loss function:

$$J(\theta) = -\log p(w_o|w_i;\theta) \tag{2.2}$$

The full formula for the loss function depends on whether we are using the CBOW or Skip-gram architecture (Rong 2014).

Word2vec employs some optimizations to increase the efficiency. *Hierarchical softmax* uses a binary tree model of the vocabulary, where the leaves are the words of the vocabulary. The probability of a word is represented by the path from the root to its leaf node. This reduces the complexity of processing a token from $O(V)$ to $O(\log_2 V)$, where V is the size of the vocabulary (Rong 2014).

Negative sampling is a simpler alternative to hierarchical softmax. It is a simplification of *noise contrastive estimation (NCE)* (Gutmann and Hyvärinen –0013–2010). Rather than updating the vectors for the whole vocabulary it uses a random subset of k samples, typically 5 to 20. The samples are drawn using a noise distribution. Lower values of k give shorter training times, and for large corpora fewer samples are needed, maybe only 2 to 5. Depending on the value of k, negative sampling can outperform hierarchical softmax both in terms of training time and word embedding quality (Mikolov, Sutskever et al. 2013; Baroni, Dinu and Kruszewski 2014). However, hierarchical softmax can give better representations of infrequent words.

The Continuous Bag-of-Word model

The CBOW model uses the context as the input of the neural network, and the target word as the output, as illustrated on the left-hand side of fig. 2.2. The network is then trained to minimize the error in predicting the target word. The resulting output weights are used as word embeddings. Calculating the weights from the input to the projection layer is straightforward. However, because of the large number of vectors that need to be updated for each token, the calculation of the weights

from the projection layer to the output layer is computationally expensive. Therefore, word2vec uses hierarchical softmax or negative sampling to increase the performance.

The skip-gram model

The skip-gram model is the opposite of the CBOW model. The target word is used as the input to the network, and the context as the output, as shown on the right-hand side of fig. 2.2. The network attempts to predict the entire context from the target word. The skip-gram model can also use hierarchical softmax or negative sampling. Because the context, which is large, is connected to the output side, which is more computationally expensive, the total complexity is greater than for CBOW. Therefore, the skip-gram model is slower than CBOW. However, this extra processing makes the skip-gram model better for infrequent words.

Performance

Word2vec has several parameters that can be tuned, for example the context window size and vector dimensionality, and most importantly the choice of model. Whether the CBOW or skip-gram model works better is task dependent, as is the choice of hyper-parameters. When implementing a system using word2vec, various combinations of model and parameters should be tested with real data in the full application.

As mentioned in section 2.2.2, frequent words can be less informative. Both CBOW and skip-gram can use subsampling of frequent words, usually above a given threshold. This ignores context words with a probability proportional to their frequencies. Not only does subsampling improve the quality of the word embeddings, it also increases performance significantly (2x-10x) because fewer tokens are processed (Mikolov, Sutskever et al. 2013). Subsampling is similar to weighting in count-based models. Subsampling also has more or less the same effect as stop word removal during linguistic preprocessing. However, it differs in that it removes most but not all tokens corresponding to the most frequent words. Therefore, stop word removal might be more efficient.

Both the CBOW and skip-gram models use bag of words contexts. They can also both skip tokens when using subsampling. Therefore, the names are not really representative of the difference between the models.

2.2.7 Word embeddings in neural networks

Neural networks are a major application of word embeddings. Word embeddings are also an important building block in neural networks for natural language processing (NLP). Neural networks can now be used for many NLP tasks (Goldberg 2016). Some common applications of neural networks are:

Neural network language models use neural networks for language modeling. N-gram language models are used to predict the next word

given a preceding sequence of words. Traditionally, this has been done by building statistical tables that are similar to co-occurrence matrices. With neural language models, a neural network is trained to perform such predictions.

Machine translation can also be done using neural networks. The task is to automatically translate text from one language into another.

Image captioning is a combination of image recognition and language processing. Here, a neural network identifies the content of a given image and produces an appropriate caption.

There are several different varieties of neural networks. The main architectures commonly used for NLP tasks are feedforward, recurrent and recursive neural networks (Goldberg 2016).

Feedforward neural networks process single input vectors, usually of fixed size. Common layers include *fully connected* and *convolutional* layers.

Recurrent neural networks (RNNs) can handle multiple inputs, like time series or sentences. The input vectors are processed sequentially. RNNs can for example be used for machine translation, where both the input and output are sequences of words.

Recursive neural networks also work with multiple input vectors. However, here the input is processed as a tree. This is suitable for instance for producing parse trees of sentences. We can also combine these network styles, for instance by using some feedforward convolutional layers to preprocess each input to a recurrent neural network.

Since neural networks are essentially mathematical functions, they operate on numbers in various forms. Their input is commonly given as vectors of floating-point numbers or *one-hot vectors*. *One-hot vectors* correspond to integers, and can be used when there is a limited number of choices, for example categories. They are binary vectors that contain zero in all places but one. This nonzero or "hot" element is one, and its position gives the corresponding integer.

Neural networks for language processing can be character-based or word-based. *Character-based* networks use letters and other characters as their basic unit. Since alphabets contain a limited number of letters, character-based networks can use one-hot vectors.

Word-based networks use words and other tokens as their basic unit. Because most application require dictionaries with at least tens of thousands of words, it would be both inefficient and impractical to use one-hot encoding of words. Instead, word-based neural networks tend to use word embeddings, typically with 100 to 1000 dimensions.

There are also some systems, like *fastText* (Bojanowski et al. 2017), which use an intermediate level of *character N-grams*. Character N-grams resemble syllables, and are especially useful for languages with a

high degree of morphology. With character N-grams, embeddings for e.g. "beginner" and "beginning" will have high similarity, because they share several N-grams. This makes better representations for compound words and inflected words.

2.2.8 Strengths, weaknesses and applications

Vector space models are well suited to calculating the similarity of arbitrary words. But because the vectors are not grouped or structured in any way, finding neighbors requires an exhaustive search. This makes applications that use the semantic neighborhood of a word computationally expensive.

Word embeddings lack interpretable dimensions. This means that while a VSM with word embeddings can tell that the words "trout" and "salmon" are quite similar, only an abstract mathematical explanation involving similarity of meaningless dimensions can be given. Sparse vectors do have meaningful dimensions, and thus can give a meaningful comparison of the words. However, they are less computationally efficient and less used. Because of the huge feature space of sparse vectors, using them in artificial neural networks is infeasible. Word embeddings on the other hand, work well with neural networks due to their low dimensionality.

Visualization is a great tool for aiding human understanding of models. Figure 2.1 shows a simple example of a graphic visualization of a vector space model. Full-scale VSMs and word vectors are hard to visualize because of their high dimensionality. Visualizing high dimensional vectors requires projecting them into a 2 or 3-dimensional space, and this can not give an accurate or faithful visualization of vectors with at least hundreds of dimensions.

There are numerous other applications of vector space models and word vectors. They can for example be used in *word sense disambiguation* (see section 5.1.1) and spelling correction. In *information retrieval* VSMs are for instance used for ranking and clustering of results. A new development is the use of word vector arithmetic. VSMs can capture relational similarities between pairs of words. For example, in the country-capital relationship, Norway:Oslo is like Denmark:Copenhagen. A well-known example is the calculation $\overrightarrow{\text{King}} - \overrightarrow{\text{Man}} + \overrightarrow{\text{Woman}} \approx \overrightarrow{\text{Queen}}$. This can be used to solve *word analogy* problems like "Smith is to iron as carpenter is to X", where task is to find X, in this case "wood". Word vectors are also somewhat composable. Vectors can be added to form new vectors representing composite words, phrases or even sentences.

2.3 Graph models

Graphs are data structures consisting of a set of *nodes* or *vertices* connected by *edges*. The edges can be directed or undirected, and weighted or unweighted. A node's *degree* is the total number of edges into or out from that node. A sparse graph has few edges, and thus a low average degree,

while a dense graph is the opposite. A graph is *complete* if each node has an edge to every other node. A *clique* is a complete *subgraph*.

A dense graph can be represented efficiently by an adjacency matrix, which for each node–node combination records the presence and possibly weight of an edge between them. For sparse graphs, most of the entries in an adjacency matrix will be zero. Therefore, it is more efficient to use adjacency lists for each node. These lists only contain nonzero values.

2.3.1 Graph distributional semantic models

Like other distributional models, graph distributional semantic models are made by processing large amounts of text. Graph models can use the same types of contexts and context features as vector models. For example, the features can be co-occurrences of terms in context windows of different sizes, or grammatical dependencies from a parsing of the context, as discussed in section 2.1.3. The JoBimText (Gliozzo et al. 2013) graph model uses a generic "holing system" to extract generalized terms called "Jos" and contexts called "Bims" from observations. The Jos may consist of multiple words, and the Bims can contain several holes where Jos occur.

Rather than using a co-occurrence matrix, each target word in the vocabulary is recorded as a node in the graph. For each node n, a list C_n of context features with occurrence counts is kept. When the corpus has been processed, these lists can be truncated or *pruned* to the top p most informative entries for each node, to increase performance. The most informative entries can be measured by frequency, possibly weighted by some kind of mutual information measure, similarly to weighting in VSMs. This pruning is somewhat like dimensionality reduction in VSMs, however the "dimensions" are local to the node/term. In word embeddings, all vectors share the same dimensions. So the total number of context features in the graph can be much greater than p (Biemann 2016).

The usual measure of the similarity between two nodes, is the number of shared context features, $\text{sim}(a, b) = \left|\{f \mid f \in C_a \wedge f \in C_b\}\right|$. This measure does not take into account the frequencies of the features. Only terms which share at least one context feature can be compared. Because the context feature lists are pruned, terms that share only infrequent, pruned features will get a similarity of zero. This contrasts to word embeddings, where even quite dissimilar words will usually get a nonzero similarity score.

The nodes in the graph are connected by weighted edges corresponding to the similarity measure. Because most terms are unrelated, the graph will be sparse. The density can be controlled by only adding edges with a minimum weight, or by limiting the outdegree of the nodes.

Apart from basic similarity, graph models can also record other relationships between nodes. Some relations, like similarity and synonymy, are *symmetric* and yield *undirected* edges. Others are *asymmetric*, for example hypernymy. They are represented by *directed* edges. JoBimText can infer synonymy and hypernymy by using *word sense induction* (see section 5.1.2), clustering and an algorithm that extracts *IS-A* relationships (hypernyms) (Gliozzo et al. 2013, p. 8). Figure 2.3 illustrates two variants

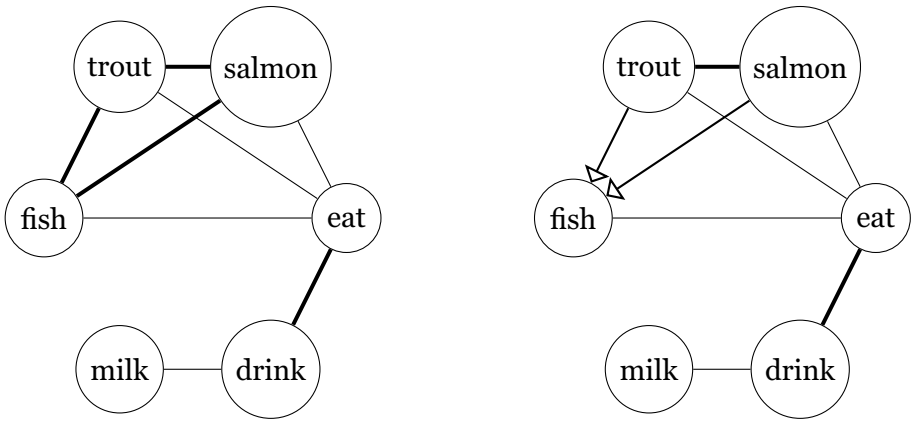


Figure 2.3: Sample GDSMs modeling similarity (left) and similarity + hypernymy (right)

of a simple graph distributional semantic model. The thickness of the lines indicate the edge weights, and arrows indicate hypernyms.

While finding neighbors in a VSM usually requires an exhaustive search of all the vectors, in a GDSM they are listed in the adjacency list. This makes graph models more efficient than vector models for some applications that entail searching for neighbors.

2.3.2 Semantic networks

Cognitive science is the interdisciplinary study of mind and thought. It includes aspects of neuroscience, computer science, linguistics and psychology, among others. Semantic networks are used in many fields, including cognitive science and natural language processing. They can represent meaning and semantic relationships on different levels, for instance in natural language. They can also be used as a model of the semantic memory structure in the brain.

Semantic networks are graphs that model various types of semantic relationships. The nodes can consist of both terms and abstract concepts or meanings. The edges represent relationships between the nodes. An edge between a term and a concept identifies the concept as one meaning of the term. There can also be edges representing other relations like synonymy, antonymy and hypernymy (Steyvers and Tenenbaum 2005).

WordNet

WordNet (Miller 1995) is a semantic network that is much used in natural language processing. It contains English content words, which are nouns, verbs, adjectives and adverbs. The words are organized into synonym sets, also called *synsets*, that are WordNet's version of concepts. There is a many-to-many relationship between words and synsets, so that each word can have multiple meanings, and each meaning or synset can have multiple members. Like traditional, printed dictionaries WordNet was

created manually by professional lexicographers. However, it was made specifically for use by computers in natural language processing. Text classification and word sense disambiguation are some of its many possible applications.

WordNet uses a number of different relationships between synsets. These include familiar ones like synonymy and antonymy, but also less commonly used relations. Hypernymy/hyponymy relates nouns with broader/narrower meaning, and troponymy is the same relation for verbs. For example, "animal" is a hypernym of "cat", and "sprint" is a troponym of "run". The hypernym/hyponym relation is also referred to as the *is-a* relation. This is convenient, for instance "salmon" *is-a* "fish". Meronymy is a relationship between a part and the whole it belongs to, like between "wheel" and "car" or between "tree" and "forest". Other relations in WordNet include entailment, similarity of words that are not synonyms and derived words.

Network structure

Semantic networks are often found to have a number of distinctive structural regularities (Steyvers and Tenenbaum 2005). The structure of the networks are typically characterized by the following properties:

Sparseness The nodes of the network have a low average degree, compared to the size of the network. Steyvers and Tenenbaum (2005) report an average degree ranging from 1.6 to 22 for different semantic networks with thousands of nodes.

Scale-free structure The degree k of the nodes in a scale-free network (Barabási and Albert 1999) follows a power law distribution, $P(k) \approx k^{-\gamma}$. Values for γ are typically in the range between 2 and 4, and for semantic networks $\gamma \approx 3$ is common. This scale-free distribution means that most nodes have only a few neighbors, while a few nodes have many. These nodes with a high degree function as hubs in the network. This is known as a *heavy-tailed* distribution, because the *tail* where $k \to \infty$ is heavier, or contains more elements, than the exponential distribution.

Small-world structure It has long been speculated that all living people can be connected by a short chain of acquaintances, "six degrees of separation". Small-world networks (Milgram 1967; Watts and Strogatz 1998) exhibit high local clustering and low average path length between nodes. This is a phenomenon found in many naturally occurring networks, like social networks.

High clustering coefficient The local clustering coefficient of a node measures how connected its neighbors are, how close they are to being a *clique*. The number of edges in a clique of size k is $\binom{k}{2} = k(k-1)/2$. The clustering coefficient C_i of a node i is given by the ratio $C_i = T_i / \binom{k_i}{2} = 2T_i / k_i(k_i - 1)$ where T_i is the total number of edges

between the k neighbors. The clustering coefficients of the nodes are averaged to find the coefficient of the graph.

DSMs and semantic networks

All graph distributional semantic models are semantic networks, since they consist of terms connected by semantic similarity. But most current GDSMs have a much simpler structure than for instance WordNet. JoBimText (see section 2.3.1) clusters terms into sets of word senses, and provides hypernyms. The rest of the relationships present in WordNet seems not to be supported yet. On the other hand, not all semantic networks are distributional. For instance, WordNet is made by hand, not by distributional methods.

Steyvers and Tenenbaum (2005) explored whether semantic networks based on vector space models have the same properties as other semantic networks, listed above. As discussed in section 3.1, there are several ways to convert a VSM to a graph model. Steyvers and Tenenbaum used a vector space model generated by Latent Semantic Analysis (LSA) (Deerwester et al. 1990), and used two different methods of converting it to a semantic network, or "LSA network". The two methods were *kNN* and a similarity threshold ε, see section 3.1.1. They found that while the LSA networks had a small-world structure, they did not display scale-free distribution. This suggests that some information is lost, either when producing the VSM or in the conversion process.

2.3.3 Visualization

Graph models represent network structures, and are therefore naturally suited to visualization. Graphs are commonly visualized similarly to fig. 2.3. Some graph models can contain different types of nodes. Semantic networks can for example consist of terms and concepts. They can also have many different kinds of relationships between the nodes. This can be illustrated using different kinds of graphics for nodes and edges.

The size of the graph is determined by the size of the vocabulary. Full-scale graph distributional semantic models will have at least thousands of nodes, usually hundreds of thousands. Since that many nodes are impossible to fit on a single page or screen, a zoom function is required. Some kind of hierarchical clustering is useful for selecting the nodes to display for different zoom levels. Hypernymy relations between words can be used if the model contains them. In the case of a flat model, a separate clustering step could be run before the visualization.

2.3.4 Strengths, weaknesses and applications

Graph distributional semantic models excel when it comes to exploring semantic neighborhoods. They work well for finding the k nearest neighbors of a word, and word sense induction. However, because the feature lists are truncated, words that only share infrequent, pruned

features will get a similarity score of zero. Therefore, GDSMs are not a good choice for calculating the similarity of arbitrary words.

Graph models can be interpreted as network structures, and are thus naturally visualized as networks like in fig. 2.3. Graph distributional semantic models are ideal for visualizing semantic relationships like synonymy and hypernymy.

Like sparse vectors, graph models keep meaningful semantic features in the model. This means that the word representations are directly interpretable, and we can make meaningful comparisons of words. However, "word arithmetic" like composition of compound words or phrases has not yet been done with GDSMs. On the other hand, graph distributional semantic models are closer than VSMs to being full semantic networks with other relationships like hypernymy. Other applications of GDSMs include word sense disambiguation and *out of vocabulary word* replacement.

2.4 Summary

There are two fundamentally different ways of building distributional semantic models: vector space models and graph-based models. While they use different representations and data structures, they also share some properties. Both types of model can be used to represent the semantics of terms, and to find the similarity of terms. They also employ some of the same methods for preprocessing observations, like weighting. While dimensionality reduction of vectors is mathematically different from pruning of context feature lists in GDSMs, the performance objective is similar. Both vector space models and graph distributional semantic models should be considered when implementing distributional semantics in an application.

The models have complimentary strengths and weaknesses. Vector space models excel at calculating word similarity and "word arithmetic", while graph models are the best choice for navigating the semantic neighborhood of words. Graph models are also the natural choice for visualizing semantic models and semantic networks. Therefore, it should be rewarding to combine the two model types. This can be done in many ways. In the next chapter, we discuss different ways of converting between vector space and graph models. We first discuss various conversion methods and their hyperparameters. Then we implement different conversion procedures.

Chapter 3

Converting between graph and vector models

In this chapter we attempt to make procedures that can convert lexical semantic vector space models to graph models, and vice versa. Our hypothesis is that lexical semantic vector space models can be converted to graph models, or vice versa, without losing significant information. We also discuss whether we should produce a complete graph, where all nodes are connected, or drop some edges. We hypothesize that our models will perform better if we drop insignificant, low similarity edges.

The conversion was implemented in Python, a high level language that is well-suited to computational science. The Python ecosystem provides a wide range of open-source libraries, including many that are useful in natural language processing. The project software is built on standard open-source tools like *Gensim* (Řehůřek and Sojka 2010) and *igraph* (Csardi and Nepusz 2006).

Igraph is an efficient graph library written in C, with the Python package *python-igraph* on top. It was chosen for the graph representation because of its ability to efficiently work with large graphs. Gensim is a Python package for working with vector space models and topic modeling.

In the next section, we discuss methods for combining or converting different lexical semantic representations. We also discuss some previous work in this field. In section 3.2, we discuss our implementation of conversion from vector space to graph models. Then, in section 3.3 we discuss our implementation of conversion from graph to vector space models. Finally, in section 3.4 we summarize this chapter.

3.1 Background and previous work

Vector space models and graph models can be combined in various ways. We can convert entire models to a different representation, or build a model that contains only an excerpt of the original model. Such an excerpt might for instance be a graph view of a neighborhood in a vector space model, or word embeddings that represent a neighborhood from a graph model.

A vector space model can be converted to a graph model by making nodes for all terms and edges based on the similarity function of the VSM. Unfortunately, conversion the other way, from a graph to a vector space model, is not so straightforward. There is no direct correspondence between the non-Euclidean geometry of graphs and Euclidean vector space models (W. L. Hamilton, R. Ying and Leskovec 2017). However, there are methods for making vectors or embeddings from graphs. In section 3.3 we discuss a direct method for making word embeddings based on the adjacency matrix of a graph model. Below, we discuss converting between vector and graph models.

3.1.1 VSM to graph conversion

Converting a vector space model to a basic graph model is quite straightforward. Ustalov et al. (2018) is a recent example that involves creating a graph model from word embeddings. However, there are some variants and hyperparameters to consider. In creating a vector space model, various processing has been applied to the raw observations from the original corpus, like weighting and dimensionality reduction. This means that the original data is not available for use in the graph model, like they would be if making a GDSM directly. The best way to convert a vector model to a graph model depends on the particulars of the VSM, like its size and dimensionality. The basic steps are to make nodes corresponding to each vector, and edges that connect similar nodes.

Nodes

If the vector model is large, it might be necessary to reduce the number of nodes that are made. This can be done by merging synonymous terms, with a similarity close to one, into a single node. This node can then store a list of the synonyms it represents, to avoid loss of information. Another method is to ignore infrequent terms. This requires sorting the terms by frequency, and discarding terms below an appropriate threshold. If the vectors in the VSM have been normalized, and do not otherwise store frequency information, this is not possible.

Vectors can also be split into multiple nodes, for example when vectors represent ambiguous terms. This can amount to *word sense induction (WSI)*, and requires some sort of semantic or mathematical analysis.

Edges

When the set of nodes is complete, edges must be added. In the rare case that the VSM happens to use a sparse representation, each vector can be used as the feature list for the corresponding graph node. In this case, regular GDSM methods can be used to complete the graph model, as described in section 2.3.1. Word embeddings might also be used as feature lists. However, because of their low dimensionality and distributed nature, they will not work well with the normal similarity measure for

graph models. This measure counts the number of features that two nodes share, but word embeddings will have nonzero values for most dimensions. Therefore, a similarity measure that takes into account the feature frequencies should be used, like with vector space models. Another option is to store the similarity values given by the vector model, usually a cosine similarity. In that case, it might not be necessary to store a feature list at all.

For performance reasons, it might be useful to limit the number of edges. The straightforward way of doing this, is by setting a minimum similarity value ε. Edges with a similarity weight lower than this threshold, are discarded. Another option is to limit the *degree* or number of edges of the nodes. This can be done by setting a limit k on the number of edges in each edge list. When constructing the graph, edge lists with more than k elements are truncated, discarding the edges with the lowest similarity scores. This effectively makes a local similarity threshold per node. Alternatively, a *kNN* algorithm can be used to limit the edges to the k *nearest neighbors*. Depending on the definition of a neighbor, this might give different results from using the k most similar nodes. For example, if it means *immediate* neighbor, two nodes are not considered neighbors if there is another node between them, as discussed next, in section 3.1.2.

3.1.2 Local graph views

Relative neighborhood graphs (RNGs) (Gyllensten and Sahlgren 2015) can be built on a VSM to explore the local properties of semantic neighborhoods. The RNG is a graph representation of the neighborhood around a given vector or point in the vector space V. Only a small part of the VSM is represented. A relative neighborhood graph is a kind of *empty region graph*, where two points a and b are considered to be neighbors if the region between them is empty. This region is defined as the intersection of two hyperspheres centered in a and b, and with radiuses equal to the distance between the two points. The relative neighborhood graph of (the endpoint of) a vector v consists of nodes corresponding to the given vector n_v, and all its neighbors n_i, with undirected edges between n_v and each n_i. Finding all the neighbors of a point requires searching through the entire VSM. For large models, this is computationally expensive. Therefore, a *kNN*-algorithm or precomputed table can be used to limit the search space. This also limits the number of neighbors found, but it should be possible to find a suitable value, for example $k = 1000$, which finds most relevant neighbors.

The neighbors of a term will often represent different senses of the term. Therefore, relative neighborhood graphs can be used for word sense induction. In order to find all the senses of a term, we must compute an RNG considering every other term in the model's vocabulary V, in other words with $k = |V|$. Gyllensten and Sahlgren (2015) calls this "global" RNG of a vector its *Semantic horizon*. This graph contains all immediate neighbors of the term.

3.1.3 Graph to VSM conversion

There are many lexical resources, like WordNet, that have a graph structure. Because VSMs are better suited to some tasks, it can be useful to convert a graph model to a vector space model. If the context lists are kept in the finished graph, these can be used to construct a co-occurrence matrix. This can then be used in a vector space model of the graph. If the lists have been pruned, this matrix will contain less information than if it had been constructed directly from the corpus. However, not many graph models have context lists. Few lexical resources come with such detailed information.

For general graph models, it is possible to make synthetic contexts by performing random walks on the graph (Piña and Johansson 2016). The nodes of a graph model are connected to nodes representing related terms. For each node in the graph G, N_{walk} random walks are performed, yielding $N_{walk} \times |G|$ random contexts, or pseudo-sentences. Each walk starts in the given node, follows random edges from node to node, and terminates with probability p_{stop} in each node. This results in contexts of variable length. Longer contexts mean that the walk will wander further away from the target term, and the context terms will be less closely related. Therefore, p_{stop} should not be too small. When a corpus of synthetic contexts have been constructed, any method can be used to make a VSM from it.

The synthetic corpus that is generated in this way, may lack some of the detail and variation that is present in natural language. However, because N_{walk} contexts are generated for each node, even infrequent terms will be well represented in the corpus.

3.1.4 Generalized node embeddings

In section 3.1.3 we discussed an NLP specific approach that built synthetic contexts by performing random walks on the graph (Piña and Johansson 2016). These contexts were then used to train a skip-gram model. There are also more general methods for creating node embeddings based on graphs.

Graph structures are usually represented by adjacency lists or an adjacency matrix. Depending on the size and density of the graph, these data structures can be large. The size of an adjacency matrix is quadratic in the number of nodes. In many machine learning applications, especially neural networks, it is preferable to work with low dimensional node embeddings.

W. L. Hamilton, R. Ying and Leskovec (2017) review the field of representation learning on graphs. There are many algorithms for making embeddings based on graphs. These can be divided into methods that make representations of *nodes*, and methods that make representations of entire *graphs* or *subgraphs*. We only discuss methods for making representations of nodes. These methods create one embedding per node, and aim to represent nodes that are close in the graph by embeddings that are close in the corresponding vector space, in other words neighboring nodes have similar embeddings (W. L. Hamilton, R. Ying and Leskovec

2017). Algorithms for learning node embeddings are often unsupervised, but supervised learning can also be used, for instance for classification tasks.

W. L. Hamilton, R. Ying and Leskovec (2017) further divide methods for making node embeddings into shallow and deep approaches. *Shallow methods* learn embeddings with no parameter sharing between nodes. Shallow methods are commonly matrix factorization based or employ random works on the graph. GraRep (Cao, Lu and Xu 2015) and HOPE (Ou et al. 2016) are examples of matrix factorization based node embedding algorithms, while DeepWalk (Perozzi, Al-Rfou and Skiena 2014) and node2vec (Grover and Leskovec 2016) use random walks.

There are two kinds of *deep methods*, autoencoders and neighborhood aggregation. *Autoencoders* are neural networks used to compress its input to a dimensionality reduced representation, z. They are made up of two parts, an encoder and a decoder. The objective is to make representations that can be decoded so that the result is as close to the input as possible, $decode(encode(x)) = decode(z) \approx x$. The autoencoder is trained to minimize the loss on a data set, in this case the graph to be encoded. The neural network learns weights that can be used to encode nodes, including new nodes that are added to the network later.

Deep Neural Graph Representations (DNGR) (Cao, Lu and Xu 2016) and Structural Deep Network Embeddings (SDNE) (D. Wang, Cui and Zhu 2016) are examples of deep methods based on autoencoders. SDNE is similar to our approach in that it uses adjacency vectors as input to the encoder. However, our approach differs in that it does not employ an autoencoder.

Neighborhood aggregation include information from neighboring nodes in the embeddings. This is done by iteratively updating the embedding for each node with the embeddings of its neighbors. Thus, each iteration increases the radius of the neighborhood that is included by one. GraphSAGE (W. Hamilton, Z. Ying and Leskovec 2017) is a neighborhood aggregation algorithm.

3.2 Converting from a vector space model to a graph model

As discussed in section 3.1.1, the basic approach to making a graph model from a vector space model is to make nodes for all the terms, and weighted edges connecting each of the nodes. However, a naïve conversion function will produce a complete graph. The number of edges in a complete graph with n nodes is $\binom{n}{2} = n(n-1)/2$. This means the space requirement of the graph is $O(n^2)$, which is impractical for large models. Additionally, a complete graph does not reflect the structure of semantic networks discussed in section 2.3.2. Semantic networks tend to be sparse, with a small-world, scale-free structure. Therefore, it is desirable to have some way to restrict the number of edges.

We investigate two different approaches here: a threshold-based

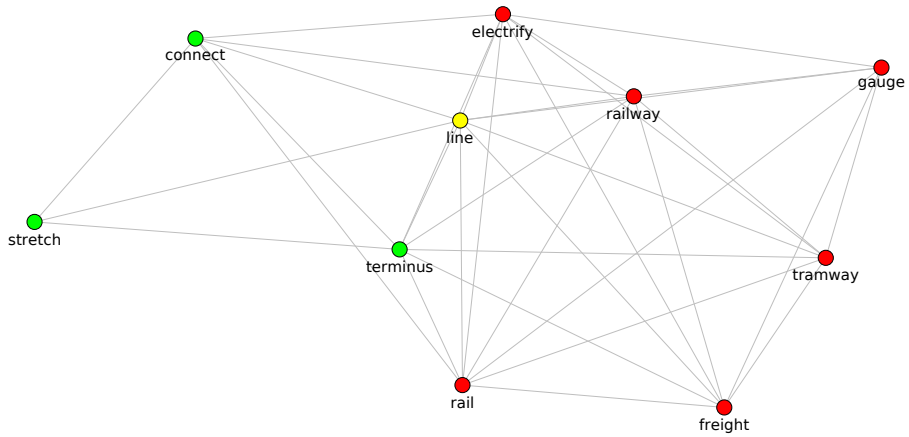


Figure 3.1: Neighborhood of "line" using threshold 0.4 in a graph with 10 000 terms. This term has quite few neighbors above this threshold. Colors indicate clusters, except yellow, which is the center.

method and a nearest neighbors method. For both methods, the starting point is a set of word embeddings. These embeddings are loaded from files stored in word2vec (Mikolov, Sutskever et al. 2013) binary format.

A graph node n_t is created for each term t in the VSM M. Thus, the set of nodes or vertices is:

$$V = \{n_t \mid t \in M\} \tag{3.1}$$

Each node is labeled with the term it represents, to enable lookup of terms in the graph. We use weighted edges. The edge weights w is the similarity of the terms the two nodes represent. Thus, an edge can be written as a triple (n_1, n_2, w), where n_i are node identifiers.

The conversion functions have different parameters. The number of terms to include is a common parameter for both methods. A lexical semantic model can contain many infrequent terms. We might not necessarily want to convert the full model, either to save time or because we do not need infrequent terms.

3.2.1 Threshold method

The threshold method uses a global similarity threshold ε. Only edges with a similarity (weight) greater than ε are added to the graph. This is an indirect limit on the number of edges, and the resulting number of edges for a given ε might vary depending on the nature of the model that is converted. We must still compute the pairwise similarity for all terms, so the amount of computation required is the same as for the naïve method. However, we end up with fewer edges, which saves space. Given a model M, and similarity function sim, the edge-set is given by:

$$E_\varepsilon = \{(n_a, n_b, sim(a,b)) \mid a, b \in M \wedge sim(a,b) \geq \varepsilon\} \tag{3.2}$$

Figure 3.1 shows the neighborhood of the term "line" from a graph constructed using the threshold method.

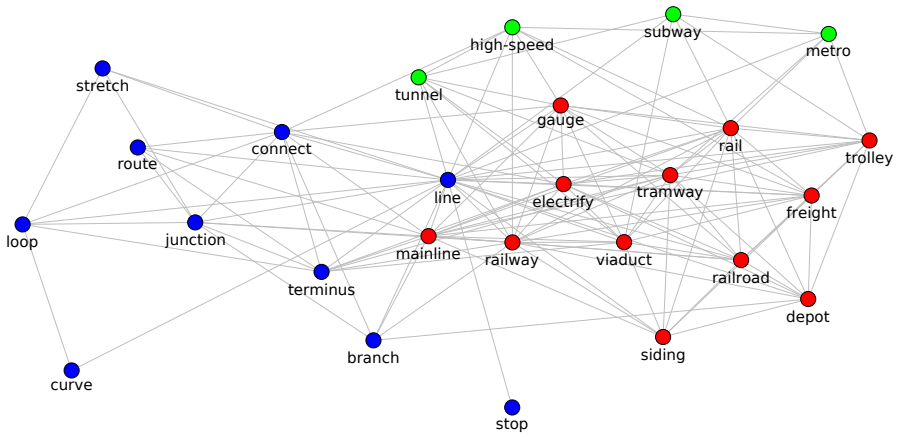


Figure 3.2: Neighborhood of "line" using $k = 25$ in a graph with 10 000 terms. Because the degree is fixed at 25, the node gets more neighbors than in fig. 3.1. Colors indicate clusters.

3.2.2 Nearest neighbors method

The nearest neighbors (kNN) method adds edges for each node to the k most similar terms, or nearest neighbors. This is somewhat similar to *kNN* classification, and requires searching through the vector space for the nearest neighbors of each term. Thus, the amount of processing is the same as for constructing a complete graph, but the space required for the resulting graph is significantly smaller. For a constant k and model M, the number of edges is $k|M|$. Thus, the space complexity is $O(|M|)$.

To find the nearest neighbors, we use the method `most_similar(term, topn = N)` provided by gensim's `KeyedVectors` class. Given a term, this method finds the top N most similar terms. The algorithm for the nearest neighbors method is similar to the one for the threshold method. The only difference is the step where the edges are added to the graph. With this method, the set of edges is defined as:

$$E_k = \{(n_a, n_b, sim(a,b)) \mid a,b \in M \wedge b \in most\_similar(a,k)\} \tag{3.3}$$

Figure 3.2 shows the neighborhood of the term "line" from a graph constructed using the kNN method. Because the degree is fixed, the node gets more neighbors than in fig. 3.1.

3.2.3 Variable-*k* method

The Variable-k method is a variation of the kNN method. As discussed in section 2.3.2 semantic networks tend to have a *scale-free* structure. However, the kNN method assigns each node a fixed number of neighbors, which does not result in a scale-free network. The vector space model we are converting probably has local structure that could be used to adjust the number of neighbors for each node. However, this would require much processing to analyze the local neighborhood of each term. We use a

simpler approach, based on the assumption that frequent terms tend to have more meanings and synonyms. Thus we can reduce k for infrequent terms. This only requires a minor modification of the kNN method, because we are already processing the terms in order of frequency. This is made possible by the fact that the word embeddings we use are ordered by frequency, as discussed in section 4.2.

We start out with k_{max} as the initial value for k. This must then be reduced by an appropriate amount for each new term. We do this by making k inversely proportional to the logarithm of the frequency rank of the term. We use the logarithm to scale down the rank because the number of terms is much larger than k_{max}. If we denote the rank of a term t by r_t and given a model M the edge-set is:

$$E_{var} = \{(n_a, n_b, sim(a,b)) \mid a, b \in M \wedge b \in most\_sim(a, k_{max}/log_{10}(10 + r_a)\}$$

The Variable-k method can also reduce the amount of memory required for large models. In a space constrained application, it is probably reasonable to store less information about less frequent terms.

3.2.4 Implementation

The conversion from vector space model to graph model is implemented by the function `VSM2graph(model, mode, threshold=-1, k=None)`. The parameter `mode` indicates the conversion method to use. `threshold` is used with the threshold method, and `k` is used with the kNN and Variable-k methods.

The function creates an igraph `Graph` object, and populates it with the terms from the VSM. Then, the edges are added according to the algorithm given above. The igraph method for adding multiple edges, `Graph.add_edges`, takes a list or generator of unweighted edges. Therefore, we first add all the edges, and then set all the weights.

3.3 Converting from a graph model to a vector space model

A graph G with N nodes can be represented by an $N \times N$ *adjacency matrix*. We only consider weighted graphs, where this matrix contains the edge weights. The matrix element $a_{i,j}$ gives the similarity (edge weight) between nodes i and j. A value of zero indicates that there is no edge connecting the nodes. This representation has much in common with a word–word co-occurrence matrix. The entries in a co-occurrence matrix are the co-occurrence frequencies of the corresponding words. The similarity scores recorded in our graph's adjacency matrix are related to these frequencies. Thus, the row for a term a in an adjacency matrix should contain much of the same information, as the row for a in a co-occurrence matrix. Therefore, it might be possible to use the adjacency matrix directly as a vector space model. This is what we attempt here.

3.3.1 Sparse matrices

As discussed, semantic networks tend to be sparse. The graph models produced by the VSM to graph conversion described in section 3.2 are also sparse to a varying degree. This means that a regular matrix representation will store many zeros. Therefore, we can save space by using a matrix representation optimized for storing only the non-zero values of sparse matrices. Reducing memory use should lead to increased performance because of better data locality. With a more economic data representation, less time is spent on transferring data between main memory and cache.

3.3.2 Dimensionality reduction

Regardless of the matrix representation used, the resulting word vectors will be high dimensional. In full-scale lexical semantic models the number of terms, N, will be upwards of 10 000. Since the adjacency matrix is square, this yields a VSM with N dimensions. Because this is impractical, we want to apply dimensionality reduction to produce word embeddings with only some hundreds of dimensions. Dimensionality reduction reduces the space needed to store the word embeddings, and this should help performance.

3.3.3 Implementation

NumPy *N-dimensional arrays (ndarrays)* are widely used for representing matrices in Python. They provide an efficient, vectorized matrix implementation written in C (Walt, Colbert and Varoquaux 2011). SciPy has several implementations of sparse matrices for different purposes. We use `scipy.sparse.lil_matrix`, which is suited to matrices with changing sparseness, where entries are changed from zero to some other value. This is required because we add edges iteratively to the matrix.

The conversion from graph to VSM is implemented by the function `graph2VSM(graph)`. This function initializes a square NumPy or SciPy matrix to the size of the graph's vocabulary. Then, it iterates over the edges in the graph, adding each one to the new matrix. In addition, every entry on the matrix diagonal represents a term's similarity to itself, and is set to one.

Scikit-learn is a Python toolkit for machine learning (Pedregosa et al. 2011). For dimensionality reduction, we apply `TruncatedSVD` from `sklearn.decomposition`. Scikit offers two different SVD algorithms, *arpack*, and a fast, *randomized* algorithm (Halko, Martinsson and Tropp 2011). While the randomized algorithm is significantly faster, it yields nondeterministic results. This leads to evaluation scores which vary between different runs. To get reproducible results we therefore use the deterministic *arpack* algorithm. In a production environment however, it would probably be preferable to use the faster randomized algorithm.

`TruncatedSVD` returns a NumPy ndarray containing the word embeddings with the specified number of features or dimensions. We use 300

dimensions, because that is the dimensionality of the pre-trained word embedding models we use for evaluation in chapter 4.

The dimensionality reduced matrix is used as the basis for a gensim `KeyedVectors` instance. This class provides various methods for working with word embeddings. However, it does not have a constructor for making an object based on an existing matrix. Therefore, making the object requires tinkering with `KeyedVectors` internals. This makes the conversion function dependent on undocumented internal details of `KeyedVectors`, but such things seem to be common in the Python world.

3.4 Summary

We have created procedures for converting lexical semantic representations. We have implemented three different methods for converting vector space models to graph models: the threshold method, the nearest neighbors method and the variable-k nearest neighbor method. We have also implemented a simple procedure for converting graph models to word embedding models. This conversion from graph to word embedding model only works with weighted graphs. All the conversion procedures were implemented in Python, using igraph for efficient graph operations.

Next, we investigate how our conversion procedures perform. In the next chapter, we evaluate the intrinsic performance of our conversion procedures. We evaluate the conversion quality using a word similarity metric, as well as the time and space use. In the following chapters, we evaluate our conversion in two different applications.

Chapter 4

Conversion evaluation by word similarity

Evaluation is an important aspect of language technology. In order to measure and compare the performance of different tools and methods, they must be evaluated somehow. Evaluation methods can be intrinsic or extrinsic.

Intrinsic evaluation means measuring the performance of a function or the quality of data in *isolation*. Usually, a function is tested on sample input, and the resulting output is compared to a *gold standard*. The gold standard is the expected, "correct" result, and is usually produced manually by a human. For example, when evaluating a part-of-speech-tagger, the sample input is a number of sentences to be tagged. The gold standard is the correct POS-tags. There exists a number of standard evaluation benchmarks and data sets for different language processing problems.

Extrinsic evaluation is measuring the performance of a function or tool within a larger system or context. This often means testing in a setting close to real-world usage. The evaluation can be based on user feedback, or automatic recording of various performance measures. In the case of a POS-tagger, we can do extrinsic evaluation by employing the tagger as part of a larger system. This might for instance be a *named entity recognizer* or a parser. We evaluate the system's performance at its task, for example named entity recognition. This requires test data, possibly annotated gold data, and an evaluation metric for this task. We keep the rest of the system fixed and change only the POS-tagger or its parameters, to isolate the effect of the changes we want to evaluate.

Extrinsic evaluation can be advantageous, because it measures the end result in a real-world application. However, it usually requires more resources and a more complicated setup than intrinsic evaluation. Therefore, intrinsic evaluation is often used in practice.

In this chapter, we evaluate the results of the conversion procedures described in chapter 3. We first use *SimLex-999*, which is a set of gold standard data for evaluating similarity scores. This is as close as we can get to intrinsic evaluation of distributional semantic models, given the absence of gold standard distributional semantic models. We may consider

word similarity evaluation as intrinsic because it is based directly on vector arithmetic. We also evaluate our results with a suite of other gold standard word similarity data from wordvectors.org.

In the next chapter, we evaluate the graph models used in an application context with SemEval-2013 task 11. That is an extrinsic evaluation task consisting of using *word sense disambiguation* and *word sense induction* to cluster search results in an information retrieval setting.

In the next section, we discuss evaluation of distributional semantic models. In section 4.2, we discuss our method for intrinsic evaluation. In section 4.3, we discuss our evaluation using SimLex-999. Section 4.4 contains our evaluation using more gold data sets from the wordvectors.org evaluation suite. In section 4.5 we summarize our intrinsic evaluation results.

4.1 Distributional semantic model evaluation

There currently exists no *gold standard* distributional semantic models or word embeddings. In fact, because many methods for generating word embeddings are nondeterministic, the same tool will generate entirely different word embeddings each time it is run on the same input. Therefore, it is probably not feasible to create gold standard word embeddings. Thus, we must evaluate the quality of distributional semantic models by how well they work in solving semantic problems.

As discussed in section 2.2.8 and section 2.3.4, there are several standard problems that can be solved using distributional semantic models. Common tasks are measuring word similarity and word analogy problems. These problems have been used as the basis for a number of benchmarks and gold standard data sets for evaluating distributional semantic models.

SimLex-999 (Hill, Reichart and Korhonen 2015) is a state-of-the-art gold standard data set for evaluating similarity scores. It consists of 999 word pairs with similarity scores averaged from ratings given by groups of human annotators. The word pairs consists of 666 noun pairs, 222 verb pairs and 111 adjective pairs. SimLex-999 emphasizes the difference between similarity and relatedness. For instance, the nouns “tea” and “cup” are highly related, since tea is served in cups. However, the two things are quite dissimilar, and have few properties in common except from being physical objects. It is therefore desirable that a distributional semantic model should be able to differentiate between similarity and relatedness.

Evaluation of similarity scores is one step away from the actual distributional semantic model, but this is currently as close as we get to intrinsic evaluation of VSMs. We might call this intrinsic evaluation, because similarity is a basic function of the semantic model, and is ordinarily only a building block of larger NLP systems.

Algorithm	NLPL id	Corpus	Vocabulary	Window
Word2vec skip-gram	11	Gigaword	261 794	5
GloVe	13	Gigaword	262 269	5
fastText skip-gram	15	Gigaword	262 269	5

Table 4.1: Models used for evaluation

4.2 Evaluating conversion results

The graph model that is produced by the VSM to graph conversion, contains the exact same similarity scores as the original vector space model yields. Evaluating the graph model therefore amounts to evaluating the performance of the original VSM. We therefore only evaluate the performance of the word embeddings produced by the graph to VSM conversion. The evaluation is done by applying both conversion functions to a model. The original VSM is first converted to a graph model, and then back to a VSM. This way, the performance of the resulting VSM can be compared to the original word embeddings.

The *NLPL word embeddings repository* (Fares et al. 2017) offers pre-computed vector space models built using a wide range of tools, algorithms and corpora.[1] We have used some of these models when evaluating the conversion results. To compare the performance of the conversion of models built using different tools, we have run the evaluation on models built with *GloVe* (Pennington, Socher and C. Manning 2014), *word2vec* (Mikolov, Sutskever et al. 2013) skip-gram and *fastText* (Bojanowski et al. 2017) skip-gram. The word2vec models are built using Gensim's word2vec implementation.

To minimize the impact of other variables, we have used models trained using similar parameters on the same corpus, the Gigaword corpus (Parker et al. 2011). All the models use untagged lemmas. The models used for evaluation are listed in table 4.1.

These models are large, with more than 200 000 terms each. Therefore, we might want to convert and evaluate only a subset of each model. The most frequent terms are probably the most relevant ones. Therefore, it would be reasonable to partition a model by frequency. Conveniently, the terms in the models are sorted by decreasing frequency. Thus, if we load the N first embeddings, we get the most frequent ones.

4.3 SimLex-999 evaluation

The VSM to graph conversion comes in three variants: the threshold-based method, the kNN method, and the variable-k method. We have evaluated all, with a wide range of parameters. The parameters to be tested are the number of terms N to include in the conversion and the method-specific edge limiting parameter ε, k, or k_{max}. Both the conversion method and

[1] http://vectors.nlpl.eu/repository/

the parameters influence the running time of the conversion as well as the performance of the resulting model.

ε is the similarity threshold, and possible values are in the range $[-1, 1]$. Most word pairs will have positive similarity, but we have included $\varepsilon = -1$ as a data point to catch any outliers with negative similarity, to get a complete model. Therefore, we have used $\varepsilon \in [-1, 0.9]$.

k is the number of neighbors each node in the graph will have. We have evaluated with $k \in [15, 1000]$. This should cover a reasonable range of values. Since semantic networks tend to be sparse, values of k much greater than 1000 are probably not helpful, and would not yield a graph model with a semantic network-like structure.

N is the model size. We have run the evaluation for increasing values of N, by extracting the N first embeddings from the evaluation model.

We have selected a single model, the *word2vec* skip-gram model, for detailed evaluation using this wide range of different parameters. That way we get a sense of the effect of the parameters. In section 4.4 we evaluate all models with a larger suite of gold standard data, but a narrower range of parameters.

4.3.1 Python implementation

While the absolute magnitudes of similarity scores can vary, we want the rank ordering of the word pair similarities to be consistent with the gold standard. Thus, *Spearman's rank order correlation* ρ is used to measure the agreement between the model's scores and the gold standard data from SimLex-999. We use `evaluate_word_pairs` from gensim's `KeyedVectors` class. It calculates the Spearman correlation between a model's similarity scores and a set of gold standard similarity ratings.

We calculate both the absolute score after conversion ρ_c and the score relative to the original score ρ_c/ρ_o. The relative score indicates how close to the original vector space model's performance we get after conversion. It is also possible to evaluate the performance of the vectors in the intermediate, square matrix. However, this score does not vary significantly from the results after dimensionality reduction. Since dimensionality reduction is not the focus here, we have not evaluated the intermediate matrix.

To compare the performance of the different conversion methods, we have also measured their execution time and memory use. We have used Python's `timeit` module for timing the different operations. For measuring memory use, we have used `Process` from the psutil[2] package. We have measured the total memory use of the Python process.

4.3.2 Threshold method

We first evaluate the threshold method for a small model, $N = 10\,000$, over the full range of possible thresholds. This model contains only the 10 000 most frequent word embeddings from our skip-gram model. Table 4.2 lists

[2] https://github.com/giampaolo/psutil

Term	Meaning
k	Number of nearest neighbors to include
k_{max}	Starting, maximum value of k for Variable-k method
ε	Similarity threshold
ρ	Spearman correlation score
ρ_o	ρ for the original model
ρ_c	ρ for the final, converted VSM
ρ_c/ρ_o	Proportion of converted to original correlation
e	Number of edges in the resulting graph
n	Number of nodes in the resulting graph
d	Density of the graph, $e/\binom{n}{2}$
t_{v2g}	Time used for the conversion from VSM to graph model, in seconds
t_{g2v}	Time used by the conversion from graph to VSM
t_r	Time used by the dimensionality reduction
m	Total memory in use by the process, in megabytes
OOV	Percentage of gold standard word pairs not present in the model (out-of-vocabulary)

Table 4.2: Notation used in the evaluation tables

the notation we use in our evaluation. The results are listed in table 4.3. As expected, the number of edges is inversely proportional to the threshold. We see that the Spearman correlation increases with the number of edges, peaking at a threshold of 0.30, with a Spearman correlation ρ_c of 0.331 and relative score ρ_c/ρ_o of 0.831. From that point, adding more edges (by lowering the threshold) degrades the performance. Table 4.3 shows that only a small fraction of the potential edges is included for the optimal threshold. For the threshold 0.30, the graph density is $d \approx 0.019$. This means that only 1.9% of the possible edges are included in the graph. Thus, we can save a significant amount of space compared to producing a complete graph. The table also shows that thresholds above 0.5 result in very low evaluation scores. Therefore, we limit further evaluation to $\varepsilon < 0.6$.

Memory and time use increase sharply as the threshold approaches -1, where the graph becomes complete. This is because the number of edges in a complete graph is quadratic in N. Thus we can expect $2N$ terms to require $4m$ memory. However, since the quality of the resulting model peaks around $\varepsilon \approx 0.3$, it is probably not desirable to compute a complete graph. For $\varepsilon \approx 0.3$ the memory requirement is smaller. To avoid unnecessary computation, we limit further evaluation to $0.2 < \varepsilon < 0.6$, and instead use smaller steps to get a more detailed picture. We next evaluate the method for $N \in \{15\,000, 20\,000, 40\,000, 80\,000, 261\,794\}$.

Evaluation scores

Tables 4.3 to 4.8 show that the max value for relative performance ρ_c/ρ_o increases with N, up to 15 000 terms. Then it more or less flattens out.

ε	ρ_c	ρ_c/ρ_o	e	d	t_{v2g} (s)	t_{g2v}	t_r	m (MB)
0.90	0.020	0.051	$1.0 \cdot 10^3$	$2.0 \cdot 10^{-5}$	1227	0	3	1810
0.80	−0.012	−0.031	$4.0 \cdot 10^3$	$8.0 \cdot 10^{-5}$	1226	0	6	1935
0.70	0.010	0.025	$9.6 \cdot 10^3$	$1.9 \cdot 10^{-4}$	1224	0	4	1959
0.60	0.121	0.303	$2.3 \cdot 10^4$	$4.5 \cdot 10^{-4}$	1226	0	4	1912
0.50	0.259	0.651	$6.1 \cdot 10^4$	$1.2 \cdot 10^{-3}$	1228	1	4	1967
0.40	0.322	0.811	$2.1 \cdot 10^5$	$4.3 \cdot 10^{-3}$	1232	3	5	2011
0.30	**0.331**	**0.831**	$9.6 \cdot 10^5$	$1.9 \cdot 10^{-2}$	1252	14	9	2198
0.20	0.299	0.751	$5.2 \cdot 10^6$	$1.0 \cdot 10^{-1}$	1372	82	34	3025
0.10	0.275	0.691	$2.7 \cdot 10^7$	$5.3 \cdot 10^{-1}$	2009	23	32	4516
−1.00	0.287	0.721	$5.0 \cdot 10^7$	1.0	2789	42	36	5886

Table 4.3: Threshold mode on 10 000 terms, 21.5% OOV, $\rho_o = 0.398$

ε	ρ_c	ρ_c/ρ_o	e	d	t_{v2g} (s)	t_{g2v}	t_r	m (MB)
0.55	0.218	0.541	$8.0 \cdot 10^4$	$7.1 \cdot 10^{-4}$	2751	1	7	1861
0.50	0.301	0.749	$1.4 \cdot 10^5$	$1.2 \cdot 10^{-3}$	2748	2	7	1881
0.45	0.336	0.835	$2.4 \cdot 10^5$	$2.2 \cdot 10^{-3}$	2718	4	8	1904
0.42	0.344	0.855	$3.4 \cdot 10^5$	$3.0 \cdot 10^{-3}$	2774	5	9	1926
0.40	0.344	0.855	$4.8 \cdot 10^5$	$4.2 \cdot 10^{-3}$	2729	7	9	1979
0.39	0.338	0.841	$5.5 \cdot 10^5$	$4.9 \cdot 10^{-3}$	2744	8	10	2004
0.38	**0.351**	**0.871**	$6.4 \cdot 10^5$	$5.7 \cdot 10^{-3}$	2737	10	11	2034
0.36	0.349	0.866	$8.6 \cdot 10^5$	$7.6 \cdot 10^{-3}$	2775	13	13	2095
0.35	0.344	0.855	$1.0 \cdot 10^6$	$8.9 \cdot 10^{-3}$	2774	15	14	2139
0.30	0.330	0.819	$2.2 \cdot 10^6$	$1.9 \cdot 10^{-2}$	2804	33	21	2336
0.25	0.312	0.774	$4.9 \cdot 10^6$	$4.4 \cdot 10^{-2}$	2870	77	40	2956

Table 4.4: Threshold mode on 15 000 terms, 9.4% OOV, $\rho_o = 0.403$

ε	ρ_c	ρ_c/ρ_o	e	d	t_{v2g} (s)	t_{g2v}	t_r	m (MB)
0.55	0.241	0.581	$1.5 \cdot 10^5$	$7.4 \cdot 10^{-4}$	4887	2	10	1894
0.50	0.309	0.745	$2.5 \cdot 10^5$	$1.2 \cdot 10^{-3}$	4879	4	10	1923
0.45	0.342	0.825	$4.4 \cdot 10^5$	$2.2 \cdot 10^{-3}$	4876	7	11	1967
0.42	0.353	0.851	$6.1 \cdot 10^5$	$3.0 \cdot 10^{-3}$	4850	9	13	2031
0.40	0.348	0.839	$8.6 \cdot 10^5$	$4.3 \cdot 10^{-3}$	4900	13	16	2112
0.39	0.348	0.840	$9.9 \cdot 10^5$	$5.0 \cdot 10^{-3}$	4918	15	16	2156
0.38	0.354	0.855	$1.1 \cdot 10^6$	$5.7 \cdot 10^{-3}$	4907	17	18	2204
0.36	**0.357**	**0.862**	$1.5 \cdot 10^6$	$7.7 \cdot 10^{-3}$	4911	24	21	2308
0.35	0.354	0.854	$1.8 \cdot 10^6$	$9.0 \cdot 10^{-3}$	4926	27	21	2370
0.30	0.337	0.812	$3.9 \cdot 10^6$	$2.0 \cdot 10^{-2}$	4986	61	37	2767
0.25	0.321	0.774	$8.8 \cdot 10^6$	$4.4 \cdot 10^{-2}$	5112	143	72	3814

Table 4.5: Threshold mode on 20 000 terms, 5.0% OOV, $\rho_o = 0.414$

ε	ρ_c	ρ_c/ρ_o	e	d	t_{v2g} (s)	t_{g2v}	t_r	m (MB)
0.55	0.249	0.593	$7.1 \cdot 10^5$	$8.9 \cdot 10^{-4}$	19873	11	25	2048
0.50	0.310	0.738	$1.1 \cdot 10^6$	$1.4 \cdot 10^{-3}$	20263	17	31	2159
0.45	0.339	0.808	$1.9 \cdot 10^6$	$2.4 \cdot 10^{-3}$	19969	28	37	2309
0.42	0.356	0.849	$2.6 \cdot 10^6$	$3.2 \cdot 10^{-3}$	20319	39	44	2520
0.40	0.351	0.835	$3.6 \cdot 10^6$	$4.5 \cdot 10^{-3}$	20238	55	56	2779
0.39	0.355	0.845	$4.1 \cdot 10^6$	$5.1 \cdot 10^{-3}$	20300	63	63	2938
0.38	**0.357**	**0.850**	$4.7 \cdot 10^6$	$5.9 \cdot 10^{-3}$	20173	73	64	2986
0.36	0.351	0.837	$6.3 \cdot 10^6$	$7.9 \cdot 10^{-3}$	20438	98	78	3378
0.35	0.350	0.835	$7.3 \cdot 10^6$	$9.1 \cdot 10^{-3}$	20227	116	87	3651
0.30	0.340	0.810	$1.6 \cdot 10^7$	$2.0 \cdot 10^{-2}$	20378	256	177	5401
0.25	0.324	0.771	$3.5 \cdot 10^7$	$4.4 \cdot 10^{-2}$	20878	608	316	9665

Table 4.6: Threshold mode on 40 000 terms, 0.9% OOV, $\rho_o = 0.420$

ε	ρ_c	ρ_c/ρ_o	e	d	t_{v2g} (s)	t_{g2v}	t_r	m (MB)
0.55	0.263	0.627	$2.6 \cdot 10^6$	$8.3 \cdot 10^{-4}$	80583	40	71	2640
0.50	0.311	0.741	$4.3 \cdot 10^6$	$1.3 \cdot 10^{-3}$	81221	67	88	3064
0.45	0.343	0.819	$7.3 \cdot 10^6$	$2.3 \cdot 10^{-3}$	83426	119	122	3742
0.42	**0.362**	**0.865**	$9.9 \cdot 10^6$	$3.1 \cdot 10^{-3}$	80625	159	153	4359
0.40	0.339	0.808	$1.4 \cdot 10^7$	$4.2 \cdot 10^{-3}$	82281	226	229	5184
0.39	0.340	0.812	$1.5 \cdot 10^7$	$4.8 \cdot 10^{-3}$	81441	255	215	5693
0.38	0.343	0.819	$1.8 \cdot 10^7$	$5.5 \cdot 10^{-3}$	80763	297	233	6184
0.36	0.344	0.822	$2.3 \cdot 10^7$	$7.2 \cdot 10^{-3}$	81238	402	350	7399
0.35	0.344	0.821	$2.7 \cdot 10^7$	$8.3 \cdot 10^{-3}$	82188	465	383	8299
0.30	0.339	0.808	$5.7 \cdot 10^7$	$1.8 \cdot 10^{-2}$	82062	1039	699	14583
0.25	0.324	0.772	$1.3 \cdot 10^8$	$4.1 \cdot 10^{-2}$	85126	2506	1394	30522

Table 4.7: Threshold mode on 80 000 terms, 0.2% OOV, $\rho_o = 0.419$

ε	ρ_c	ρ_c/ρ_o	e	d	t_{v2g} (s)	t_{g2v}	t_r	m (MB)
0.55	0.210	0.501	$4.4 \cdot 10^7$	$1.3 \cdot 10^{-3}$	936897	762	752	11892
0.50	0.254	0.606	$7.2 \cdot 10^7$	$2.1 \cdot 10^{-3}$	864718	1287	1184	17504
0.45	0.269	0.642	$1.2 \cdot 10^8$	$3.5 \cdot 10^{-3}$	874222	2235	1718	27329
0.40	**0.288**	**0.689**	$2.1 \cdot 10^8$	$6.1 \cdot 10^{-3}$	891863	3933	2878	45251

Table 4.8: Threshold mode on 261 794 terms, 0.2% OOV, $\rho_o = 0.419$

The best relative score is achieved with $\varepsilon = 0.38$ and $N = 15000$. For this configuration, the relative performance ρ_c / ρ_o is 0.871. The best absolute performance is attained with the largest model, $N = 80000$. Here, the threshold $\varepsilon = 0.42$ yields a Spearman correlation $\rho_c = 0.362$.

Time and space use

Table 4.7 shows that the time use for models with 80 000 terms is substantial. The majority of the time is taken up by the conversion from word embeddings to graph model, t_{v2g}. This step takes more than 80 000 seconds, or 22 hours. For the full model with 261 794 terms, shown in table 4.8, the conversion takes 864 718 seconds, or around 10 days. This time is independent of the threshold, since all possible pairs of nodes must be examined to determine if their similarity is above the threshold. Therefore, the threshold method is very time-consuming for large models. The memory requirement is still moderate for 80 000 terms, with the optimal threshold of $\varepsilon = 0.42$ using only around 4 GB memory. The full model however, requires around 17 GB memory for the conversion, with a high threshold of 0.5.

4.3.3 kNN method

We evaluate the performance of the kNN method for $N \in \{10\,000, 15\,000, 20\,000, 40\,000, 80\,000, 261\,794\}$. 261 794 is the size of the full word embedding model. The results are listed in tables 4.9 to 4.14. The kNN method outperforms the threshold method both in terms of relative score ρ_c / ρ_o, execution time and memory use.

The kNN method gives higher scores with far fewer edges than the threshold method. This might be because the kNN method ensures that all nodes in the graph are connected to k neighbors, regardless of their similarity. In contrast, the threshold method can result in some nodes having very few neighbors. A constant number of neighbors might yield a better representation of the neighborhoods, which could explain the superior correlation scores of this method. Another possible explanation, is that the edges with a high similarity score need not be the most characteristic.

Time and space use

The kNN method is much more efficient than the threshold method. The conversion of a model with 80 000 terms takes only around 900 seconds, or 15 minutes. In contrast, the threshold method used more than 22 hours for a model of this size. The main reason for this improved performance is the efficient lookup of the k nearest neighbors in the word embedding model provided by gensim. Table 4.14 shows that we can convert the full word embedding model in 2 to 3 hours. The memory requirement for converting the full model varies with k, from around 5 GB for low values of k with low evaluation scores, to around 30 GB for larger values with higher scores.

k	ρ_c	ρ_c/ρ_o	e	d	t_{v2g} (s)	t_{g2v}	t_r	m (MB)
15	0.356	0.895	$1.0 \cdot 10^5$	$2.1 \cdot 10^{-3}$	6	2	4	1759
20	0.364	0.915	$1.4 \cdot 10^5$	$2.8 \cdot 10^{-3}$	7	2	4	1894
25	0.367	0.923	$1.7 \cdot 10^5$	$3.4 \cdot 10^{-3}$	8	3	5	1922
30	**0.369**	**0.927**	$2.0 \cdot 10^5$	$4.1 \cdot 10^{-3}$	9	3	5	1877
40	0.364	0.916	$2.7 \cdot 10^5$	$5.4 \cdot 10^{-3}$	11	4	5	1958
50	0.365	0.917	$3.3 \cdot 10^5$	$6.7 \cdot 10^{-3}$	13	5	5	1990
60	0.358	0.900	$4.0 \cdot 10^5$	$8.0 \cdot 10^{-3}$	15	6	6	1949
70	0.353	0.888	$4.6 \cdot 10^5$	$9.2 \cdot 10^{-3}$	17	7	6	2031
100	0.353	0.888	$6.5 \cdot 10^5$	$1.3 \cdot 10^{-2}$	23	10	7	1967
150	0.346	0.870	$9.6 \cdot 10^5$	$1.9 \cdot 10^{-2}$	33	14	9	2087
200	0.338	0.850	$1.3 \cdot 10^6$	$2.5 \cdot 10^{-2}$	43	19	11	2256
300	0.329	0.826	$1.9 \cdot 10^6$	$3.8 \cdot 10^{-2}$	61	28	15	2400

Table 4.9: kNN mode on 10 000 terms, 21.5% OOV, $\rho_o = 0.398$

k	ρ_c	ρ_c/ρ_o	e	d	t_{v2g} (s)	t_{g2v}	t_r	m (MB)
20	0.362	0.900	$2.1 \cdot 10^5$	$1.9 \cdot 10^{-3}$	12	3	8	1839
25	**0.374**	**0.928**	$2.6 \cdot 10^5$	$2.3 \cdot 10^{-3}$	13	4	9	1861
30	0.371	0.922	$3.1 \cdot 10^5$	$2.8 \cdot 10^{-3}$	15	5	9	1882
40	0.372	0.924	$4.1 \cdot 10^5$	$3.7 \cdot 10^{-3}$	18	6	10	1922
50	0.370	0.920	$5.1 \cdot 10^5$	$4.6 \cdot 10^{-3}$	21	7	10	1960
60	0.370	0.920	$6.1 \cdot 10^5$	$5.4 \cdot 10^{-3}$	24	9	11	1955
70	0.369	0.917	$7.1 \cdot 10^5$	$6.3 \cdot 10^{-3}$	27	10	12	2035
100	0.363	0.902	$1.0 \cdot 10^6$	$8.9 \cdot 10^{-3}$	36	14	14	2120
150	0.352	0.874	$1.5 \cdot 10^6$	$1.3 \cdot 10^{-2}$	50	21	17	2277
200	0.351	0.871	$1.9 \cdot 10^6$	$1.7 \cdot 10^{-2}$	65	28	20	2396
300	0.340	0.845	$2.9 \cdot 10^6$	$2.6 \cdot 10^{-2}$	92	42	26	2623

Table 4.10: kNN mode on 15 000 terms, 9.4% OOV, $\rho_o = 0.403$

Evaluation scores

The best relative score $\rho_c/\rho_o \approx 0.928$ is achieved at $k = 25$ with $\rho_c = 0.374$ for one of the smaller models, with $N = 15\,000$. However, the best absolute score is obtained with the configuration $k = 50$ and $N = 20\,000$, with Spearman correlation $\rho_c = 0.382$. From there, increasing the model size slightly decreases the evaluation score. This might be due to noise introduced by infrequent terms. The quality of low-frequency terms or vectors may be lower because of data sparseness. We try to alleviate this by using a variable value for k next.

k	ρ_c	ρ_c/ρ_o	e	d	t_{v2g} (s)	t_{g2v}	t_r	m (MB)
20	0.380	0.916	$2.9 \cdot 10^5$	$1.4 \cdot 10^{-3}$	18	4	12	1899
25	0.379	0.914	$3.5 \cdot 10^5$	$1.8 \cdot 10^{-3}$	20	5	12	1933
30	0.378	0.912	$4.2 \cdot 10^5$	$2.1 \cdot 10^{-3}$	22	6	12	1959
40	0.379	0.915	$5.6 \cdot 10^5$	$2.8 \cdot 10^{-3}$	26	8	13	2001
50	**0.382**	**0.921**	$6.9 \cdot 10^5$	$3.5 \cdot 10^{-3}$	30	10	14	2044
60	0.378	0.911	$8.3 \cdot 10^5$	$4.1 \cdot 10^{-3}$	35	12	15	2104
70	0.372	0.898	$9.6 \cdot 10^5$	$4.8 \cdot 10^{-3}$	39	14	16	2173
100	0.370	0.893	$1.4 \cdot 10^6$	$6.8 \cdot 10^{-3}$	51	20	19	2225
150	0.362	0.872	$2.0 \cdot 10^6$	$1.0 \cdot 10^{-2}$	73	29	25	2484
200	0.357	0.862	$2.6 \cdot 10^6$	$1.3 \cdot 10^{-2}$	92	39	30	2690
300	0.354	0.854	$3.9 \cdot 10^6$	$1.9 \cdot 10^{-2}$	133	57	37	2924

Table 4.11: kNN mode on 20 000 terms, 5.0% OOV, $\rho_o = 0.414$

k	ρ_c	ρ_c/ρ_o	e	d	t_{v2g} (s)	t_{g2v}	t_r	m (MB)
30	0.369	0.880	$8.8 \cdot 10^5$	$1.1 \cdot 10^{-3}$	102	13	29	2201
40	0.369	0.880	$1.2 \cdot 10^6$	$1.5 \cdot 10^{-3}$	109	17	32	2291
50	0.371	0.884	$1.5 \cdot 10^6$	$1.8 \cdot 10^{-3}$	121	21	35	2395
60	0.374	0.892	$1.7 \cdot 10^6$	$2.2 \cdot 10^{-3}$	127	26	38	2519
70	0.376	0.897	$2.0 \cdot 10^6$	$2.5 \cdot 10^{-3}$	136	30	41	2510
100	**0.378**	**0.901**	$2.8 \cdot 10^6$	$3.6 \cdot 10^{-3}$	151	42	49	2910
150	0.369	0.879	$4.2 \cdot 10^6$	$5.3 \cdot 10^{-3}$	208	64	68	3009
200	0.365	0.869	$5.6 \cdot 10^6$	$6.9 \cdot 10^{-3}$	262	85	75	3235
300	0.361	0.861	$8.2 \cdot 10^6$	$1.0 \cdot 10^{-2}$	331	124	98	3855
400	0.359	0.855	$1.1 \cdot 10^7$	$1.3 \cdot 10^{-2}$	422	164	118	4502
600	0.355	0.845	$1.6 \cdot 10^7$	$2.0 \cdot 10^{-2}$	584	243	195	5740

Table 4.12: kNN mode on 40 000 terms, 0.9% OOV, $\rho_o = 0.420$

k	ρ_c	ρ_c/ρ_o	e	d	t_{v2g} (s)	t_{g2v}	t_r	m (MB)
30	0.338	0.807	$1.8 \cdot 10^6$	$5.7 \cdot 10^{-4}$	658	29	79	2683
40	0.347	0.829	$2.4 \cdot 10^6$	$7.6 \cdot 10^{-4}$	399	37	97	2944
50	0.351	0.839	$3.0 \cdot 10^6$	$9.4 \cdot 10^{-4}$	418	46	87	3144
60	0.350	0.836	$3.6 \cdot 10^6$	$1.1 \cdot 10^{-3}$	469	55	94	3333
70	0.351	0.838	$4.2 \cdot 10^6$	$1.3 \cdot 10^{-3}$	488	65	99	3513
100	0.360	0.860	$5.9 \cdot 10^6$	$1.8 \cdot 10^{-3}$	516	92	113	3578
150	**0.363**	**0.867**	$8.8 \cdot 10^6$	$2.7 \cdot 10^{-3}$	612	138	149	4231
200	0.359	0.856	$1.2 \cdot 10^7$	$3.6 \cdot 10^{-3}$	701	182	182	4923
300	0.356	0.851	$1.7 \cdot 10^7$	$5.4 \cdot 10^{-3}$	890	273	243	6160
400	0.354	0.844	$2.3 \cdot 10^7$	$7.0 \cdot 10^{-3}$	1056	362	306	7482
600	0.353	0.842	$3.3 \cdot 10^7$	$1.0 \cdot 10^{-2}$	1438	541	399	10041

Table 4.13: kNN mode on 80 000 terms, 0.2% OOV, $\rho_o = 0.419$

k	ρ_c	ρ_c/ρ_o	e	d	t_{v2g} (s)	t_{g2v}	t_r	m (MB)
50	0.270	0.646	$1.0 \cdot 10^7$	$3.0 \cdot 10^{-4}$	3555	158	335	5726
60	0.271	0.646	$1.2 \cdot 10^7$	$3.6 \cdot 10^{-4}$	3603	189	367	6221
70	0.272	0.650	$1.4 \cdot 10^7$	$4.2 \cdot 10^{-4}$	3653	222	400	6711
100	0.272	0.649	$2.0 \cdot 10^7$	$6.0 \cdot 10^{-4}$	3878	318	496	8006
150	0.290	0.692	$3.0 \cdot 10^7$	$8.8 \cdot 10^{-4}$	4173	468	547	10233
200	0.297	0.708	$4.0 \cdot 10^7$	$1.2 \cdot 10^{-3}$	4533	636	687	12630
300	0.304	0.726	$5.9 \cdot 10^7$	$1.7 \cdot 10^{-3}$	5184	943	951	16920
400	0.311	0.742	$7.8 \cdot 10^7$	$2.3 \cdot 10^{-3}$	5806	1262	1325	21553
600	**0.314**	**0.749**	$1.2 \cdot 10^8$	$3.4 \cdot 10^{-3}$	7060	1891	1930	29972
1000	0.313	0.747	$1.9 \cdot 10^8$	$5.5 \cdot 10^{-3}$	9612	3157	2978	46383

Table 4.14: kNN mode on 261 794 terms, 0.2% OOV, $\rho_o = 0.419$

4.3.4 Variable-*k* method

We evaluate the Variable-k method using the same values for N as for the kNN method. The parameter to this method is the starting value k_{max}. The value of k decreases sharply for the first terms, to $k_{max}/2$ for term 90, $k_{max}/3$ for term 990 and $k_{max}/4$ for term 9990. This means that we must use k_{max} approximately 3-4 times the k we would use with regular, static kNN. The results are listed in tables 4.15 to 4.20. They show that the Variable-k method scores somewhat better than the kNN method.

Evaluation scores

The Variable-k method achieves the best relative score with the smallest model, where $N = 10000$. Here, $k_{max} = 80$ yields $\rho_c/\rho_o = 0.944$ and $\rho_c = 0.375$. For the full model with all 261 794 terms, the Spearman correlation is $\rho_c = 0.339$. This is the best result for the full model, higher than the correlation of 0.314 for the kNN method.

Time and space use

The Variable-k method retains less information for infrequent terms. This saves memory, possibly at the cost of accuracy for rare terms. However, SimLex-999 contains few rare words. The evaluation data report 5% out of vocabulary words when evaluating the 20 000 most frequent words from the word2vec skip-gram model. This means that 95% of the SimLex-999 terms are among the 20 000 most frequent words. Therefore, evaluation with SimLex-999 does not cover the remaining around 200 000 terms in the model. This means that SimLex-999 will not give a good indication of the quality of the infrequent terms in the model. Thus, we are unable to measure the accuracy cost of the Variable-k method for infrequent words.

k_{max}	ρ_c	ρ_c/ρ_o	e	d	t_{v2g} (s)	t_{g2v}	t_r	m (MB)
20	0.339	0.852	$4.6 \cdot 10^4$	$9.1 \cdot 10^{-4}$	4	1	6	1750
40	0.351	0.883	$8.4 \cdot 10^4$	$1.7 \cdot 10^{-3}$	5	1	6	1879
60	0.362	0.910	$1.2 \cdot 10^5$	$2.5 \cdot 10^{-3}$	7	2	5	1775
80	**0.375**	**0.944**	$1.6 \cdot 10^5$	$3.2 \cdot 10^{-3}$	8	2	5	1910
100	0.370	0.930	$2.0 \cdot 10^5$	$4.0 \cdot 10^{-3}$	9	3	5	1938
120	0.365	0.919	$2.4 \cdot 10^5$	$4.7 \cdot 10^{-3}$	10	3	5	1894
160	0.361	0.909	$3.1 \cdot 10^5$	$6.2 \cdot 10^{-3}$	12	5	5	1972
200	0.356	0.894	$3.9 \cdot 10^5$	$7.7 \cdot 10^{-3}$	15	6	6	1880
240	0.349	0.878	$4.6 \cdot 10^5$	$9.2 \cdot 10^{-3}$	17	7	6	2026
280	0.347	0.874	$5.3 \cdot 10^5$	$1.1 \cdot 10^{-2}$	19	8	7	1925

Table 4.15: Variable-k mode on 10 000 terms, 21.5% OOV, $\rho_o = 0.398$

k_{max}	ρ_c	ρ_c/ρ_o	e	d	t_{v2g} (s)	t_{g2v}	t_r	m (MB)
20	0.333	0.827	$6.4 \cdot 10^4$	$5.7 \cdot 10^{-4}$	8	1	10	1790
40	0.362	0.900	$1.2 \cdot 10^5$	$1.1 \cdot 10^{-3}$	9	2	9	1811
60	0.372	0.923	$1.8 \cdot 10^5$	$1.6 \cdot 10^{-3}$	11	3	8	1832
80	**0.378**	**0.938**	$2.3 \cdot 10^5$	$2.1 \cdot 10^{-3}$	12	4	8	1858
100	0.377	0.938	$2.9 \cdot 10^5$	$2.6 \cdot 10^{-3}$	14	4	8	1875
120	0.378	0.938	$3.4 \cdot 10^5$	$3.1 \cdot 10^{-3}$	16	5	9	1897
160	0.375	0.931	$4.5 \cdot 10^5$	$4.0 \cdot 10^{-3}$	19	7	9	1942
200	0.368	0.915	$5.6 \cdot 10^5$	$5.0 \cdot 10^{-3}$	23	8	10	1945
240	0.364	0.904	$6.7 \cdot 10^5$	$6.0 \cdot 10^{-3}$	26	10	11	2029
280	0.362	0.900	$7.8 \cdot 10^5$	$6.9 \cdot 10^{-3}$	29	12	12	2061

Table 4.16: Variable-k mode on 15 000 terms, 9.4% OOV, $\rho_o = 0.403$

k_{max}	ρ_c	ρ_c/ρ_o	e	d	t_{v2g} (s)	t_{g2v}	t_r	m (MB)
20	0.368	0.887	$8.4 \cdot 10^4$	$4.2 \cdot 10^{-4}$	12	1	16	1816
40	0.374	0.901	$1.6 \cdot 10^5$	$8.0 \cdot 10^{-4}$	13	2	13	1850
60	0.377	0.910	$2.3 \cdot 10^5$	$1.2 \cdot 10^{-3}$	15	4	13	1877
80	**0.388**	**0.937**	$3.1 \cdot 10^5$	$1.5 \cdot 10^{-3}$	18	5	12	1909
100	0.381	0.919	$3.8 \cdot 10^5$	$1.9 \cdot 10^{-3}$	20	6	12	1941
120	0.386	0.932	$4.5 \cdot 10^5$	$2.3 \cdot 10^{-3}$	23	7	12	1970
160	0.383	0.923	$6.0 \cdot 10^5$	$3.0 \cdot 10^{-3}$	26	9	13	2020
200	0.381	0.920	$7.4 \cdot 10^5$	$3.7 \cdot 10^{-3}$	30	11	14	2080
240	0.377	0.910	$8.8 \cdot 10^5$	$4.4 \cdot 10^{-3}$	35	13	15	2135
280	0.372	0.898	$1.0 \cdot 10^6$	$5.1 \cdot 10^{-3}$	42	15	16	2172

Table 4.17: Variable-k mode on 20 000 terms, 5.0% OOV, $\rho_o = 0.414$

k_{max}	ρ_c	ρ_c/ρ_o	e	d	t_{v2g} (s)	t_{g2v}	t_r	m (MB)
40	0.358	0.853	$3.1 \cdot 10^5$	$3.8 \cdot 10^{-4}$	86	5	28	1991
60	0.367	0.874	$4.5 \cdot 10^5$	$5.6 \cdot 10^{-4}$	88	7	31	2069
80	0.372	0.886	$5.9 \cdot 10^5$	$7.4 \cdot 10^{-4}$	93	9	32	2119
100	0.369	0.879	$7.3 \cdot 10^5$	$9.1 \cdot 10^{-4}$	97	11	34	2156
120	0.372	0.887	$8.7 \cdot 10^5$	$1.1 \cdot 10^{-3}$	102	13	35	2227
160	**0.377**	**0.899**	$1.1 \cdot 10^6$	$1.4 \cdot 10^{-3}$	111	17	31	2322
200	0.377	0.899	$1.4 \cdot 10^6$	$1.8 \cdot 10^{-3}$	119	22	39	2408
240	0.375	0.893	$1.7 \cdot 10^6$	$2.1 \cdot 10^{-3}$	124	25	38	2497
280	0.377	0.899	$2.0 \cdot 10^6$	$2.5 \cdot 10^{-3}$	139	29	40	2621
400	0.376	0.896	$2.8 \cdot 10^6$	$3.5 \cdot 10^{-3}$	163	43	56	2751
600	0.366	0.871	$4.1 \cdot 10^6$	$5.2 \cdot 10^{-3}$	210	63	70	3042

Table 4.18: Variable-k mode on 40 000 terms, 0.9% OOV, $\rho_o = 0.420$

k_{max}	ρ_c	ρ_c/ρ_o	e	d	t_{v2g} (s)	t_{g2v}	t_r	m (MB)
40	0.321	0.766	$6.0 \cdot 10^5$	$1.9 \cdot 10^{-4}$	354	10	74	2296
60	0.328	0.784	$8.7 \cdot 10^5$	$2.7 \cdot 10^{-4}$	348	14	77	2419
80	0.334	0.797	$1.1 \cdot 10^6$	$3.6 \cdot 10^{-4}$	361	18	75	2510
100	0.339	0.810	$1.4 \cdot 10^6$	$4.4 \cdot 10^{-4}$	376	21	77	2616
120	0.336	0.801	$1.7 \cdot 10^6$	$5.3 \cdot 10^{-4}$	376	26	78	2708
160	0.350	0.836	$2.2 \cdot 10^6$	$6.9 \cdot 10^{-4}$	395	34	89	2877
200	0.356	0.849	$2.8 \cdot 10^6$	$8.6 \cdot 10^{-4}$	409	42	81	3068
240	0.352	0.839	$3.3 \cdot 10^6$	$1.0 \cdot 10^{-3}$	465	51	84	3192
280	0.353	0.842	$3.8 \cdot 10^6$	$1.2 \cdot 10^{-3}$	450	58	100	3412
400	0.361	0.862	$5.4 \cdot 10^6$	$1.7 \cdot 10^{-3}$	524	82	108	3438
600	**0.362**	**0.864**	$8.0 \cdot 10^6$	$2.5 \cdot 10^{-3}$	594	123	141	4049

Table 4.19: Variable-k mode on 80 000 terms, 0.2% OOV, $\rho_o = 0.419$

k_{max}	ρ_c	ρ_c/ρ_o	e	d	t_{v2g} (s)	t_{g2v}	t_r	m (MB)
40	0.337	0.806	$1.8 \cdot 10^6$	$5.3 \cdot 10^{-5}$	3282	29	284	3670
60	**0.339**	**0.810**	$2.7 \cdot 10^6$	$7.9 \cdot 10^{-5}$	3304	42	261	3965
80	0.328	0.782	$3.5 \cdot 10^6$	$1.0 \cdot 10^{-4}$	3405	54	300	4237
100	0.323	0.772	$4.4 \cdot 10^6$	$1.3 \cdot 10^{-4}$	3393	67	294	4402
120	0.291	0.694	$5.2 \cdot 10^6$	$1.5 \cdot 10^{-4}$	3392	79	302	4552
160	0.280	0.669	$6.9 \cdot 10^6$	$2.0 \cdot 10^{-4}$	3460	105	332	4894
200	0.289	0.691	$8.5 \cdot 10^6$	$2.5 \cdot 10^{-4}$	3511	130	306	5299
240	0.284	0.678	$1.0 \cdot 10^7$	$3.0 \cdot 10^{-4}$	3562	156	334	5693
280	0.285	0.680	$1.2 \cdot 10^7$	$3.4 \cdot 10^{-4}$	3606	179	358	6084
400	0.283	0.677	$1.7 \cdot 10^7$	$4.9 \cdot 10^{-4}$	3785	253	436	7143
600	0.288	0.688	$2.5 \cdot 10^7$	$7.2 \cdot 10^{-4}$	4058	381	473	8972

Table 4.20: Variable-k mode on 261 794 terms, 0.2% OOV, $\rho_o = 0.419$

Name	Word pairs	Reference
MC-30	30	Miller and Charles (1991)
MEN	3000	Bruni et al. (2012)
MTurk-287	287	Radinsky et al. (2011)
MTurk-771	771	Halawi et al. (2012)
RG-65	65	Rubenstein and Goodenough (1965)
Rare-Word	2034	Luong, Socher and C. Manning (2013)
SimLex-999	999	Hill, Reichart and Korhonen (2015)
Verb-143	143	Baker, Reichart and Korhonen (2014)
WS-353	353	Finkelstein et al. (2002)
WS-353-REL	252	Agirre, Alfonseca et al. (2009)
WS-353-SIM	203	Agirre, Alfonseca et al. (2009)
YP-130	130	Yang and Powers (2006)

Table 4.21: Faruqui and Dyer (2014) gold standard evaluation data

4.4 Wordvectors.org evaluation suite

Wordvectors.org is a website for evaluation and exchange of vector space models (Faruqui and Dyer 2014). It provides an evaluation suite consisting of evaluation software and several sets of gold standard evaluation data[3]. We use only the gold standard data sets here, not the evaluation software. For consistency, we continue to use gensim for evaluation. Otherwise, we might get slightly different evaluation results with gensim and the wordvectors.org evaluation suite due to different evaluation implementations. Table 4.21 lists the gold standard similarity data sets.

To compare the performance of the conversion of word embeddings built with different tools, we evaluate the results of converting three different vector space models. The models are made with word2vec skip-gram, GloVe and fastText, and are listed in table 4.1.

Each model is evaluated with both the kNN and Variable-k methods. For both methods the full word embedding models are converted to graph models and then back to vector models. We do not evaluate the threshold method here, because converting the full models with the threshold method would be excessively time-consuming. We use the parameters that achieved the highest SimLex-999 evaluation score in the previous section. For the kNN method, this is $k = 600$, while for Variable-k we use $k_{max} = 60$. We evaluate the resulting, converted VSM as well as the original model.

Tables 4.22 to 4.23 list the resulting Spearman correlation scores. They suggest that the conversion performs well with all three different tools used to generate the models. Results for the converted models are mostly within 20-30% of the scores for the original models. The worst performance is with the Verb-143 data set. For this gold data set, all the models get a Spearman correlation score close to zero. However, this also applies to the

[3] https://github.com/mfaruqui/eval-word-vectors

Dataset	word2vec ρ_o	word2vec ρ_c	fastText ρ_o	fastText ρ_c	GloVe ρ_o	GloVe ρ_c
MC-30	0.647	0.676	0.667	0.696	0.704	0.496
MEN	0.604	0.539	0.617	0.558	0.608	0.477
MTurk-287	0.682	0.599	0.692	0.640	0.601	0.552
MTurk-771	0.587	0.480	0.611	0.501	0.591	0.442
RG-65	0.605	0.558	0.611	0.544	0.688	0.483
Rare-Word	0.489	0.388	0.499	0.388	0.407	0.352
SimLex-999	0.419	0.314	0.412	0.316	0.377	0.273
Verb-143	0.069	0.028	0.039	0.029	−0.071	−0.117
WS-353-REL	0.591	0.500	0.611	0.514	0.536	0.436
WS-353-SIM	0.688	0.550	0.697	0.571	0.590	0.471
WS-353	0.646	0.517	0.655	0.534	0.556	0.438
YP-130	0.545	0.230	0.551	0.278	0.506	0.262

Table 4.22: Evaluation suite on full model, using the kNN method

Dataset	word2vec ρ_o	word2vec ρ_c	fastText ρ_o	fastText ρ_c	GloVe ρ_o	GloVe ρ_c
MC-30	0.647	0.543	0.667	0.446	0.704	0.338
MEN	0.604	0.437	0.617	0.451	0.608	0.467
MTurk-287	0.682	0.472	0.692	0.497	0.601	0.537
MTurk-771	0.587	0.356	0.611	0.373	0.591	0.442
RG-65	0.605	0.451	0.611	0.326	0.688	0.363
Rare-Word	0.489	0.351	0.499	0.317	0.407	0.311
SimLex-999	0.419	0.339	0.412	0.305	0.377	0.313
Verb-143	0.069	0.040	0.039	−0.011	−0.071	−0.084
WS-353-REL	0.591	0.357	0.611	0.428	0.536	0.456
WS-353-SIM	0.688	0.403	0.697	0.372	0.590	0.527
WS-353	0.646	0.384	0.655	0.408	0.556	0.489
YP-130	0.545	0.249	0.551	0.240	0.506	0.291

Table 4.23: Evaluation suite on full model, using Variable-*k*

original models, which suggests that the origin of the poor performance is in the original data. The variable-*k* method scores best on the SimLex-999 evaluation for two of the three models. Variable-*k* outperforms kNN for the word2vec and GloVe models, while kNN scores best for the fastText model.

The tables also contain the results for the Rare-Words gold data set. This is a collection of 2034 word pairs, where at least one of the terms in each pair is infrequent (Luong, Socher and C. Manning 2013). We use this data set to evaluate the performance of our converted models for infrequent terms. Table 4.22 shows the scores for the models converted with the kNN method. The converted word2vec, fastText and GloVe models get Spearman correlation scores ρ_c of 0.388, 0.388 and 0.352, respectively, for the Rare-Words data set. This is higher than the scores for the models converted with the Variable-*k* method, listed in table 4.23. With the Variable-*k* method, the scores are 0.351, 0.317 and 0.311, respectively. This matches our expectation that the Variable-*k* method will have lower performance than the kNN method for infrequent terms. The reason for this, is that the Variable-*k* method keeps less information for rare words.

4.5 Summary

We have performed an intrinsic evaluation of our conversion procedures for lexical semantic representations. The evaluation of the conversion results was done using SimLex-999 and several other gold standard data sets from the wordvectors.org evaluation suite. We have evaluated three different methods for converting from vector space models to graph models: the threshold method, the kNN method and the variable-k method. Our evaluation shows that all of these methods are suitable for converting vector space models to graph models.

The methods have different benefits and drawbacks that makes them suitable for different problems. The threshold method is most likely to produce graphs with a small-world structure. Some algorithms works only with small-world graphs. This includes HyperLex (Véronis 2004), which we discuss in section 5.1.3. For the threshold method, we obtain the best evaluation scores with thresholds in the range between 0.2 and 0.5. The kNN method outperforms the threshold method in terms of both evaluation scores and efficiency. The optimal value for k increases with the size of the model to be converted, from 25 with 15 000 terms 600 for the full model. However, the variable-k method scores best on the SimLex-999 evaluation for two of the three word embedding models we use. Variable-k outperforms kNN for the word2vec and GloVe models, while kNN has the highest score for the fastText model.

Our evaluation results show that the conversion procedures work quite well. The Spearman correlation score of the converted word embeddings is generally about 80% of the score for the original word embedding models. The best score is above 94% of the original score. This score is nearly perfect, thus most of the information from the original model is retained. In other words, the conversion is nearly lossless. This supports our hypothesis that lexical semantic representations can be converted without significant loss of information. Our results also support our hypothesis that it is beneficial to drop some edges. The evaluations of all three methods show that the performance increases, in terms of both quality and computing cost, when insignificant edges are dropped. In the next two chapters, we show that this level of performance also results in good performance in two applications.

We have also evaluated our conversion from a graph model to a word embedding model. All our evaluation scores are for the full evaluation, converting from word embeddings to a graph model, and back to word embeddings. Therefore, some of the loss can be attributed to each conversion step. Our results suggest that the conversion loss is small in both conversion steps.

In this chapter, we have evaluated word embeddings. These were obtained by converting word embedding models first to a graph model, and then back to word embeddings. Therefore, we also implicitly evaluated the intermediate graph representation. In the next chapter, we evaluate the graph model itself. This is done as extrinsic evaluation by means of a word sense induction task from SemEval-2013.

Chapter 5

Conversion evaluation by word sense induction

In this chapter, we perform extrinsic evaluation of our converted graph models. Extrinsic evaluation means evaluating some function or data as part of a larger system. Extrinsic evaluation is usually closer to real-world use than intrinsic evaluation. Often, extrinsic evaluation in the end-user application will give the most realistic results. However, such evaluation can be expensive. We evaluate the performance of our graph models when employed in word sense induction (WSI) and word sense disambiguation (WSD) in SemEval-2013 task 11.

Task 11 (Navigli and Vannella 2013) of the SemEval-2013 workshop on semantic evaluation was an extrinsic evaluation task. It evaluated WSI and WSD systems in the context of web search. Participating systems were given 100 ambiguous web search queries with corresponding sets of 64 search results for each query.[1] The search queries contained one to four words, either single words or multiword expressions. The search results contained the title, URL and a text snippet from each page. The systems were required to cluster each result set according to the senses of the search query. With a WSI system, this can be done by first inducing senses for the query, and then use these senses to cluster the results.

We have used task 11 for extrinsic evaluation of the converted graph model. This entails using a graph-based word sense induction algorithm to induce a set of senses based on our graph. We have attempted to use both the HyperLex and spinglass (Reichardt and Bornholdt 2006) graph clustering algorithms for word sense induction and matching of search snippets to senses.

Our hypothesis is that a graph model with *low similarity* edges filtered out will perform better at word sense induction. The assumption is that low similarity edges contributes little useful information for this task. To test this hypothesis, we have evaluated doing WSI on graph models produced with different similarity thresholds.

In the next section, we discuss word senses. We discuss word sense induction and disambiguation in general, and HyperLex in particular. In

[1] https://www.cs.york.ac.uk/semeval-2013/task11/index.php%3Fid=data.html

section 5.2 we discuss our implementation of word sense induction and disambiguation. We have employed two methods for word sense induction: HyperLex and spinglass clustering. In section 5.3 we discuss our method for evaluating our word sense induction and disambiguation. Section 5.4 contains the evaluation results, and in section 5.5 we do an error analysis. Finally, in section 5.6 we summarize our results in this chapter.

5.1 Word senses

Individual word semantics is an important building block in sentence semantics. To be able to understand the meaning of the text, we must first understand the words it consists of. Words can have many different meanings or *senses*. *Homonyms* are entirely different words that are spelled or pronounced the same. For instance, the word "bank" can mean a financial institution or the land beside a river. Words with different but closely related meanings are known as *polysemous*. The polysemous word "card" can refer to a playing card or credit card. There is no clear-cut line between homonymy and *polysemy*, but a gradual transition. Different senses of a word are usually denoted by a number. The different senses of "bank" can be called $bank^1$, $bank^2$ and so on.

Dictionaries and thesauruses are well-known sources of information about word senses in everyday life. In natural language processing we use lexical resources like WordNet. WordNet groups words with similar meaning into synonym sets or *synsets*. Words that have different meanings are linked to the synsets that represent each of the meanings.

5.1.1 Word sense disambiguation

Homonymy and polysemy are important factors in the ambiguity of natural language. In NLP tasks where we need some understanding of the semantics of text, we might need to disambiguate it. *Word sense disambiguation* is the task of determining which sense a type or token of text refers to. Similarly to part-of-speech tagging, it is commonly done by tagging ambiguous tokens with its sense. For instance, the sentence "I like to fish" could be tokenized and word sense annotated as $[I, like/like^1, to, fish/fish^3]$.

There are two standard WSD tasks: *all words* and *lexical sample* WSD. A lexical sample is a short fragment of text with only a single or a few target words to be disambiguated. It could for instance be a search string from an information retrieval system. Normally, a set of senses is provided only for the target word(s). This makes the search space quite small. In all words WSD the goal is to disambiguate all polysemantic words in a longer text. This problem is similar to part-of-speech tagging. However, part-of-speech tagging uses a small set of perhaps 50 tags, while in WSD each word can typically have five to ten senses. In a generic system with a dictionary containing at least 10 000 words, there can be more than 100 000 senses.

This makes word sense disambiguation much more complex than part-of-speech tagging.

There are three main approaches to word sense disambiguation (Navigli 2012). *Supervised* WSD is trained on a corpus that is manually annotated with word senses. Supervised classifiers include memory based methods and SVM classifiers. Creating handmade corpora is expensive and time-consuming. This limits the use of supervised WSD.

Knowledge-based WSD builds on existing knowledge resources like lexicons, thesauruses and ontologies. WordNet is an important knowledge resource, but many different lexicons and domain specific resources exists. A common approach is to use graph based methods that explore a graph built on for instance WordNet. Another well-known method is the Lesk algorithm (Lesk 1986). This algorithm measures the overlap between the context of the word to be disambiguated and dictionary definitions of its senses.

Unsupervised WSD uses unlabeled corpora to automatically learn words senses. This is also known as *word sense induction*. Word sense induction can be used to automatically build sense inventories from a corpus of text.

WSD Evaluation

Word sense disambiguation is usually not a goal in itself, but is used as a processing step in a larger system that requires semantic information. As such, WSD should ideally be evaluated extrinsically as part of the intended application. However, it is sometimes necessary to evaluate word sense disambiguation intrinsically. Intrinsic evaluation is commonly done with manually annotated gold standard test data. The word sense disambiguation is run on an untagged version of the test data, and the resulting tags are compared to the gold standard tags. If the disambiguation is allowed to skip ambiguous words, precision and recall can be calculated. Otherwise, a simple accuracy score is sufficient.

The *Senseval* project, later renamed *SemEval* has made several resources for WSD evaluation (Kilgarriff and Palmer 2000). They include test problems and gold standard data for intrinsic evaluation of word sense disambiguation. SemEval also has several tasks involving text analysis that can be used as extrinsic evaluation of a WSD component as part of a larger system.

5.1.2 Word sense induction

Word sense induction bypasses the need for manually created resources like tagged corpora, lexicons or other knowledge resources. Different word senses are induced from regular text. WSI can be used to make lexical resources similar to WordNet from large text corpora. This can be general resources, or domain specific resources built from domain specific corpora. Word sense induction can also use smaller fragments of text, for instance to cluster search results by topic (Navigli and Crisafulli 2010).

Most forms of word sense induction use some kind of clustering based on the distributional hypothesis. This hypothesis states that words with similar meaning tend to occur in similar contexts. The idea is to use this to find words with similar semantics. The clustering generally gathers words with similar meaning into synsets. There are several variations:

Context clustering uses the context of each word, represented as a *context vector*, often just a bag of words. These contexts are clustered into synsets.

Word clustering also clusters words by their meaning, but use different representations from context clustering. Here, syntactic contexts are often used.

Co-occurrence graphs builds graph based on the co-occurrence of context features. The graphs are analyzed to find synsets. One example is HyperLex (Véronis 2004). HyperLex analyzes graphs to find hubs at the center of high-density components.

Probabilistic clustering uses probabilistic methods to generate context distributions for different senses of a word.

Topic modeling approaches use topic modeling methods like *Latent Dirichlet Allocation (LDA)* for clustering.

WSI Evaluation

Word sense induction is a less mature field than word sense disambiguation. This is also the case with evaluation of WSI. A good way to evaluate word sense induction is extrinsic evaluation as part of a larger system. However, this is not always possible or desirable. In such cases, we must use intrinsic evaluation. Since word sense induction essentially is a clustering problem, this amounts to evaluation of the resulting clustering. Evaluation of clustering is a difficult problem that is often done using annotated test data. This is unfortunate, since avoiding hand-annotating data is one of the objectives of word sense induction. However, there are some resources for WSI evaluation. The SemEval project included word sense induction tasks in several of its iterations (Navigli and Vannella 2013; Manandhar et al. 2010).

Several different evaluation strategies have been tried, but a standard approach for WSI evaluation has yet to emerge. Navigli and Vannella (2013) evaluated clustering of search results. The results were clustered according to the induced senses of the search query. They used four different measures of clustering quality based on a gold standard clustering:

- Rand index (RI) (Rand 1971)
- Adjusted Rand index (ARI) (Hubert and Arabie 1985)
- Jaccard index (Jaccard 1901)

- F1 measure (van Rijsbergen 1979)

All these measures are also relevant for intrinsic evaluation of word sense induction. However, they require handmade gold standard clusters as a basis of comparison.

5.1.3 HyperLex

HyperLex is a word sense induction algorithm originally intended for clustering results in information retrieval (Véronis 2004). It exploits the *small-world* structure of co-occurrence graphs. Small-world graphs (see section 2.3.2) have clusters of highly connected nodes, and thus a high clustering coefficient. These clusters are also connected, giving a short average path length between arbitrary nodes.

While the HyperLex algorithm was originally intended for use in information retrieval, it can also be generalized to use regular corpora (Agirre, Martínez et al. 2006).

Construction

HyperLex does *unsupervised* word sense induction based on an untagged text corpus. The basic HyperLex algorithm does word sense induction for a single word. A separate co-occurrence graph is constructed for each target term. Each co-occurrence graph is specific to this search term or *target* word, and is used to disambiguate this term. Therefore, the algorithm first makes a subcorpus specific to the target term. This subcorpus consists of all sentences containing the target term.

Stop words and infrequent terms are removed from the subcorpus, and a co-occurrence matrix is made, similarly to traditional vector space models. The co-occurrence matrix is used to construct a co-occurrence graph. The edges in the graph have weights in the interval $[0,1]$. The weight indicate the distance between the two words it connects. Thus, completely synonymous terms are connected by an edge with weight 0. This is the opposite measure of the well-known word similarity. The edge weight between nodes a and b is $w_{a,b} = 1 - max[P(a \mid b), P(b \mid a)]$ where $P(a \mid b) = f_{a,b} / f_b$.

Once the graph has been constructed, the algorithm uses it to find clusters or *high-density* components. These components are taken to represent word senses. Clusters are identified by their center nodes, or hubs, which have a high degree. Candidate hubs are nodes with degree and clustering coefficient over given thresholds. The algorithm selects the node with the highest degree in the graph. If this node is a candidate hub, it is assumed to represent a word sense. The hub is added to the set of word senses, and it and all its immediate neighbors are "removed" from the graph. This is usually done by tagging them as used. Now, the next word sense can be identified by the node which currently has the highest degree. This procedure is repeated until no candidate hubs remain.

Since frequent terms tend to co-occur with many different terms, they are likely to have many neighbors. Hence, the node degree can be approximated by its term frequency for efficiency.

When all word senses have been identified, all nodes are arranged into a tree. The root of the tree is the target term, and the cluster hubs are connected as its children with edge weights zero. These children of the root node are the induced senses. The rest of the tree is constructed by making a minimum spanning tree using the rest of the nodes. This way, each node is attached to the cluster hub closest to it in the original graph.

Disambiguation

The minimum spanning tree is used to make vectors that represent its terms. The vectors have as many dimensions or components as the target word has senses. All the terms are given a score vector containing zeros in all but one position. This position is equal to the sense number, and the value indicates the degree of association between the term and the sense.

For instance, when disambiguating the target term "bank", the context term "river" could be represented by the vector $(0.0, 0.0, 0.79)$. This indicates that the target term has three different senses, and that "river" is associated with the third sense. The component values are in the range $[0, 1]$, where high values mean a close association with that sense. Formally, a term t is represented by a vector v, with components given by

$$v_i = \begin{cases} \frac{1}{1+d(s_i,t)}, & \text{if } t \text{ is a descendent of sense } s_i \\ 0, & \text{otherwise} \end{cases} \tag{5.1}$$

Here, $d(s_i, t))$ is the weighted distance from the sense hub s_i to the term node t.

Disambiguation is done on a target word in a small context, for example a sentence or a search snippet. All the terms in the context of the target word are collected, and their vectors are summed. The resulting vector represents collective "meaning" of the context, and the largest component is the sense most strongly associated with this meaning. A reliability score can also be calculated, based on the difference between the components of the vector sum.

5.2 Implementation

We do word sense induction on a graph model by graph clustering or community detection. We evaluate the effectiveness of both HyperLex and spinglass community detection for WSI. Since graph clustering is sensitive to the density of the graph, the number of edges influence the performance. Hence, the similarity threshold used to construct or convert the graph is an important hyperparameter. To be able to efficiently cluster the graph, we extract a local *subgraph* from the original *source* graph.

We use the page title and snippet from each search result for word sense induction and disambiguation. These data are provided as text strings,

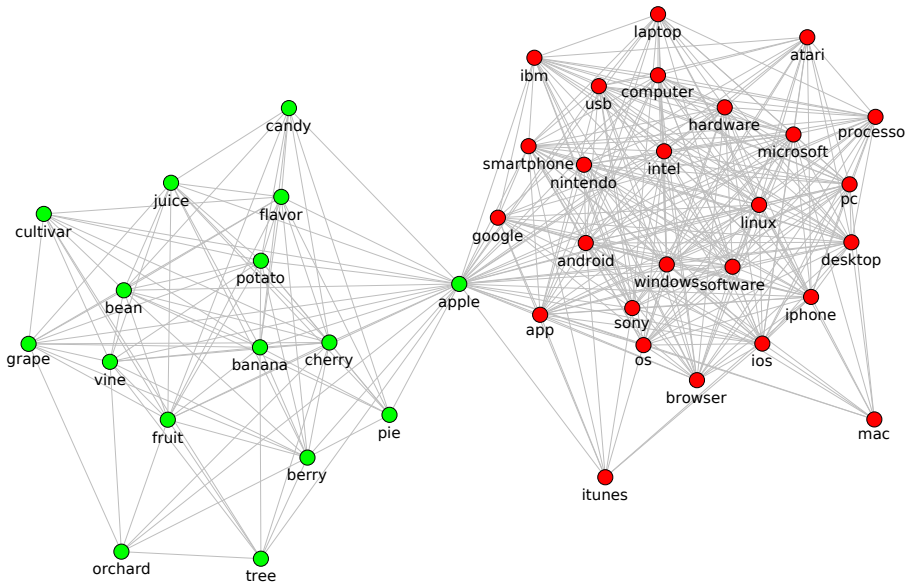


Figure 5.1: Neighborhood of "apple" using threshold 0.4 in a graph with 10 000 terms. The colors illustrate clusters.

so we use *Stanford CoreNLP* (C. Manning et al. 2014) to tokenize and lemmatize the text. We use the same CoreNLP options that were used to generate our word embedding model. These options are *"tokenize, ssplit, pos, lemma, ner"*, which also enables part-of-speech tagging and named entity recognition *(NER)*. We do not employ the POS-tags here. We must lemmatize the text to match the embeddings in our original vector space model.

5.2.1 Graph model

The basis for the word sense induction is a graph model converted from a word embedding model as described in section 3.2. We make this graph model from word embeddings trained on a Wikipedia corpus. Graph-based word sense induction algorithms like HyperLex tend to exploit the small-world structure of semantic graphs, as discussed in section 5.1.3. The kNN method yields graphs where the node degree is fixed at k, which is definitely not a small-world structure. This makes the kNN method ill suited for this task.

Figure 5.1 and fig. 5.2 illustrate the difference between the threshold method and the kNN method. Figure 5.1 shows the neighborhood of "apple" in a graph produced using the threshold method. The nodes cluster nicely into two groups, representing the *computer manufacturer* and *fruit* senses of "apple". In contrast, fig. 5.2 plots the neighbors of "apple" in a graph produced using the kNN method. Although in the full graph produced by the kNN method, all nodes have degree 25, in this subgraph the degree varies somewhat. This is because edges to nodes outside the neighborhood are not included. Nevertheless, this graph seems to be harder to cluster

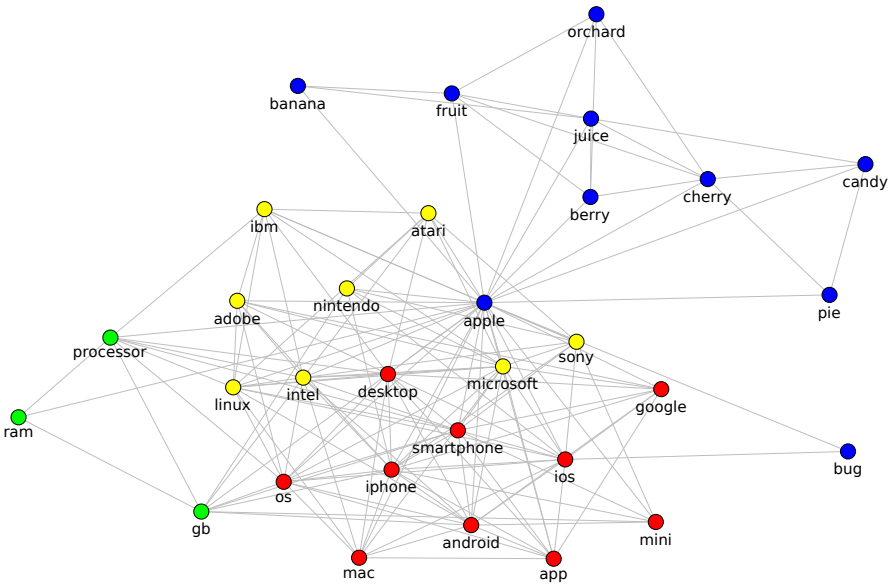


Figure 5.2: Neighborhood of "apple" using $k = 25$ in a graph with 10 000 terms. Because all nodes have a similar degree, this subgraph is harder to cluster then the one in fig. 5.1.

than the one produced using the threshold method. We therefore make a graph based on the threshold method. We create the largest feasible graph to encompass as many of the terms from SemEval-2013 task 11 as possible. This yields a graph containing the 220 000 most frequent terms from the VSM, with a threshold of 0.4.

Task 11 contains named entities and other multiword expressions that are currently not recognized by the CoreNLP named entity recognizer, like "Romeo and Juliet". Since this is not an evaluation of NER systems, we overcome this problem by manually gluing together all the multiword expressions from task 11, so that for instance "bald eagle" becomes "bald::eagle" throughout the corpus. This means that we must use a purpose built vector space model for this task. We train a word2vec skip-gram model on our modified Wikipedia corpus. Still, rare expressions like "the::wizard::of::oz" are missing from the model, due to data sparsity. Table 5.1 lists the terms from task 11 that do not exist in our graph model. There are 21 of these out-of-vocabulary (OOV) terms.

5.2.2 Neighborhood graph extraction

We want to do word sense induction on the neighborhood of the target term to be disambiguated. To do this, we must first extract the relevant neighborhood as a separate subgraph. We therefore collect all terms from all the contexts given for the target. We select the nodes corresponding to these context terms as well as the target term. These nodes, and the edges connecting them, make up the subgraph we use for word sense induction. Thus, the same graph is used to disambiguate the target in all contexts.

OOV term
ten::commandments
brett::butler
courtney::cox
babel::fish
the::last::supper
romeo::and::juliet
medal::of::honor
arch::of::triumph
dead::or::alive
man::in::black
heaven::and::hell
stand::by::me
prince::of::persia
billy::the::kid
sense::and::sensibility
soldier::of::fortune
beauty::and::the::beast
lord::of::the::flies
battle::of::the::bulge
the::da::vinci::code
the::wizard::of::oz

Table 5.1: SemEval-2013 task 11 terms that are not in our graph model

Of course, this is only possible when all contexts to be disambiguated are known when we perform the word sense induction.

Since the density of a graph influences graph clustering, it is beneficial to have a means of adjusting it. The base threshold is set when the source graph is constructed, and any edges with similarity below this threshold are lost. Therefore, we are unable to lower the threshold. However, we can raise the threshold by dropping edges from the graph. Thus, we include the similarity threshold as a parameter to the WSI procedure.

Some context terms might not be connected to the rest of the graph. This means that they form separate small or singleton clusters. These small clusters should probably most often not be interpreted as separate senses of the target word. In addition, some clustering algorithms, like spinglass, require connected graphs to work. Therefore, we remove disconnected nodes by only using the largest connected component of the subgraph. When the subgraph to be used for word sense induction is constructed, we apply one of the following algorithms.

5.2.3 HyperLex

HyperLex is implemented as described in Véronis (2004), see section 5.1.3. Because our lexical semantic model has already been converted to a graph representation, we can skip the graph construction step of HyperLex. Thus, we start out with the clustering step of HyperLex, and build a minimum spanning tree. HyperLex tends to produce many small clusters, which give many small "micro-senses".

We build a HyperLex instance for each query string to be disambiguated. This instance is then used to disambiguate the query sense for each search result. The resulting clustering is written to a file on the format specified by the task 11 evaluator software, see section 5.3. The evaluator is then run on the results file to obtain the evaluation scores.

5.2.4 Igraph spinglass community detection

Igraph implements several community detection algorithms. Here, we apply the spinglass algorithm (Reichardt and Bornholdt 2006). Word sense induction consists of running the spinglass algorithm on the neighborhood subgraph described above. Spinglass produces a clustering of all the nodes in the subgraph, where each cluster represents a sense.

Spinglass clustering can be done with or without using the edge weights. With complete or high-density graphs, the edge weights are needed to successfully cluster the graph. The weights also contain information that can be useful when clustering sparser graphs. Therefore, we use spinglass with edge weights enabled. Because spinglass is a stochastic algorithm, the result can differ between different executions on the same graph.

The spinglass algorithm has a parameter for limiting the maximum number of clusters. Such a limit would be useful if we had some other way of determining the number of word senses before running the clustering. However, we have no way of knowing this number a priori. Determining the

number of senses is indeed part of the problem of determining the senses of a word, which we are trying to solve.

Each of the clusters produced by spinglass is interpreted as a sense of the target word. When the clustering is complete, we perform word sense disambiguation based on these clusters or senses. We disambiguate the meaning of the search result snippets. For each cluster, we count the number of tokens from the text snippet that belongs to it. The cluster with the highest count is selected as the induced sense.

5.2.5 Alternative, on-the-fly conversion

The word sense induction procedure described above uses a complete graph model, which in our case was converted from a word embedding model. We can call this a pre-converted graph model. For large models, this conversion can be time-consuming, up to several days for a model with around 200 000 terms. An alternative approach is to construct only the required neighborhood graph or regional graph directly from the word embedding model. This *on-the-fly* conversion is done by building a list of context words as described in section 5.2. Then, a new graph is built containing only nodes for the target and context words. We add edges to this graph by the regular threshold method, as described in section 3.2.1. To find an appropriate value for the threshold parameter, we run the evaluation for different values of the threshold. When the neighborhood graph has been constructed, we use the same method as before for WSI and WSD.

Such an on-the-fly converted neighborhood graph should be isomorphic to a subgraph with the same terms made from a graph based on the same VSM. Say we use word embedding model E to create graph model G using threshold ε. Further, we have a list of terms t. We then create a subgraph S of G consisting of the terms in t and the edges connecting them. We also make a graph U from E containing the terms in t using on-the-fly conversion with threshold ε. Now, both S and U both consists of nodes corresponding to the terms in t. Both graphs also have the same edges between the nodes based on the similarities in E, using the threshold ε. Since S and U have exactly the same nodes and edges, they are isomorphic.

For low thresholds, the number of edges increases towards $\binom{n}{2} = n(n-1)/2$. This makes building large graph models for thresholds near zero impractical. For tasks requiring such low thresholds, this on-the-fly approach would be more efficient.

5.3 Evaluation

The word sense disambiguation yields a clustering of the search results. We evaluate the clustering results with the evaluation software provided for SemEval-2013 task 11.[2] The evaluation suite consists of a Java program

[2] https://www.cs.york.ac.uk/semeval-2013/task11/data/uploads/semeval-2013_task11_evaluator.tar.gz

and gold standard clustering data.[3] The program computes the correlation between our clustering and the gold standard clustering. It outputs several different scores, including Rand Index, Adjusted Rand Index (ARI), F1 and Jaccard Index. We focus on the ARI, since this score corrects for incidental correlation.

We evaluate our word sense induction and disambiguation with three different lexical semantic models. First, we evaluate WSI and WSD on a graph model converted from a word embedding model with threshold 0.4. This graph model contains 220 000 terms. As discussed in section 5.2.1 this word embedding model is custom built to include the multi-word expressions that occur in SemEval-2013 task 11. We evaluate HyperLex and spinglass word sense induction with thresholds ranging from 0.4 to 0.54. We also evaluate both WSI methods using on-the-fly conversion from the same word embedding model. Because of the smaller memory requirement of on-the-fly conversion, we are able to evaluate this method with thresholds ranging from zero to 0.54. We evaluate two different variations of on-the-fly conversion. First, we limit the number of terms to the size of the full, pre-converted graph model, 220 000 terms. This means the results should be comparable to the pre-converted graph model. Next, we do an evaluation based on the full 634 999 terms in the word embedding model. That way, we use the full vocabulary of our word embedding model. To sum up, we perform our evaluation with three different models:

1. A pre-converted graph model with 220 000 terms.
2. A word embedding model with 220 000 terms, using on-the-fly conversion.
3. A word embedding model with 634 999 terms, using on-the-fly conversion.

We evaluate word sense induction and disambiguation with HyperLex and spinglass on each of the three models. This makes six different configurations in all.

Because the spinglass algorithm is stochastic, the results might vary between different executions with the same graph. We therefore run spinglass five times for each configuration of hyperparameters, and calculate the mean and standard deviation of the ARI scores.

5.4 Results

Table 5.2 lists the results of our evaluation in bold. The results of SemEval-2013 task 11 are included for comparison, taken from Navigli and Vannella (2013). The table lists the optimal threshold for each method, and the corresponding Adjusted Rand Index score. Spinglass clustering with on-the-fly conversion scores best of our methods. Our system scores

[3] https://www.cs.york.ac.uk/semeval-2013/task11/data/uploads/datasets/semeval-2013_task11_dataset.tar.gz

System	ARI	Threshold	Model size
HDP-CLUSTERS-NOLEMMA	21.49	–	–
HDP-CLUSTERS-LEMMA	21.31	–	–
Spinglass, on-the-fly, full model	10.44	0.42	634 999
Spinglass, on-the-fly	9.73	0.36	220 000
Spinglass	9.36	0.44	220 000
HyperLex, on-the-fly, full model	8.23	0.24	634 999
HyperLex, on-the-fly	8.18	0.36	220 000
HyperLex	7.57	0.42	220 000
SATTY-APPROACH1	7.19	–	–
DULUTH.SYS7.PK2	6.78	–	–
DULUTH.SYS1.PK2	5.74	–	–
UKP-WSI-WP-LLR2	3.77	–	–
UKP-WSI-WP-PMI	3.64	–	–
DULUTH.SYS9.PK2	2.59	–	–
UKP-WSI-WACKY-LLR	2.53	–	–

Table 5.2: Results of SemEval-2013 task 11, from Navigli and Vannella (2013). Our results are added in bold.

higher than seven of the nine systems that participated in the WSI part of SemEval-2013 task 11. If we account for the fact that the two best performers, HDP-CLUSTERS-LEMMA and HDP-CLUSTERS-NOLEMMA, were variations of the same system, our system ranks second in this list.

Figure 5.3 plots the ARI scores for different thresholds, with error bars that indicate the standard deviation. Table 5.3 lists the time used. The times include both WSI and WSD on the 100 target terms in the data set. Overall, we achieve the best scores with thresholds in the interval from 0.2 to 0.5.

The plot in fig. 5.3 shows that scores for on-the-fly conversion are more or less the same as the scores for WSI based on a full graph model of the same size. This is as expected, since the neighborhood graph should end up the same whether the selection of terms is done before or after the graph conversion. Nevertheless, the optimal thresholds in table 5.2 are different, because on-the-fly conversion is evaluated over a wider range of thresholds. The plot shows that there is some variation in the results for the same threshold, which is probably due to the nondeterministic nature of the spinglass algorithm. HyperLex can also be somewhat nondeterministic, in cases where two nodes are tied for the highest degree.

5.4.1 Spinglass

We obtain the best score with the largest model, as might be expected. The best score for the full model is around 10.4%, with threshold 0.42. The best score for the smaller model is around 9.7%, and is obtained with a threshold of 0.36. Interestingly, the evaluation score increases by only 0.7 percentiles

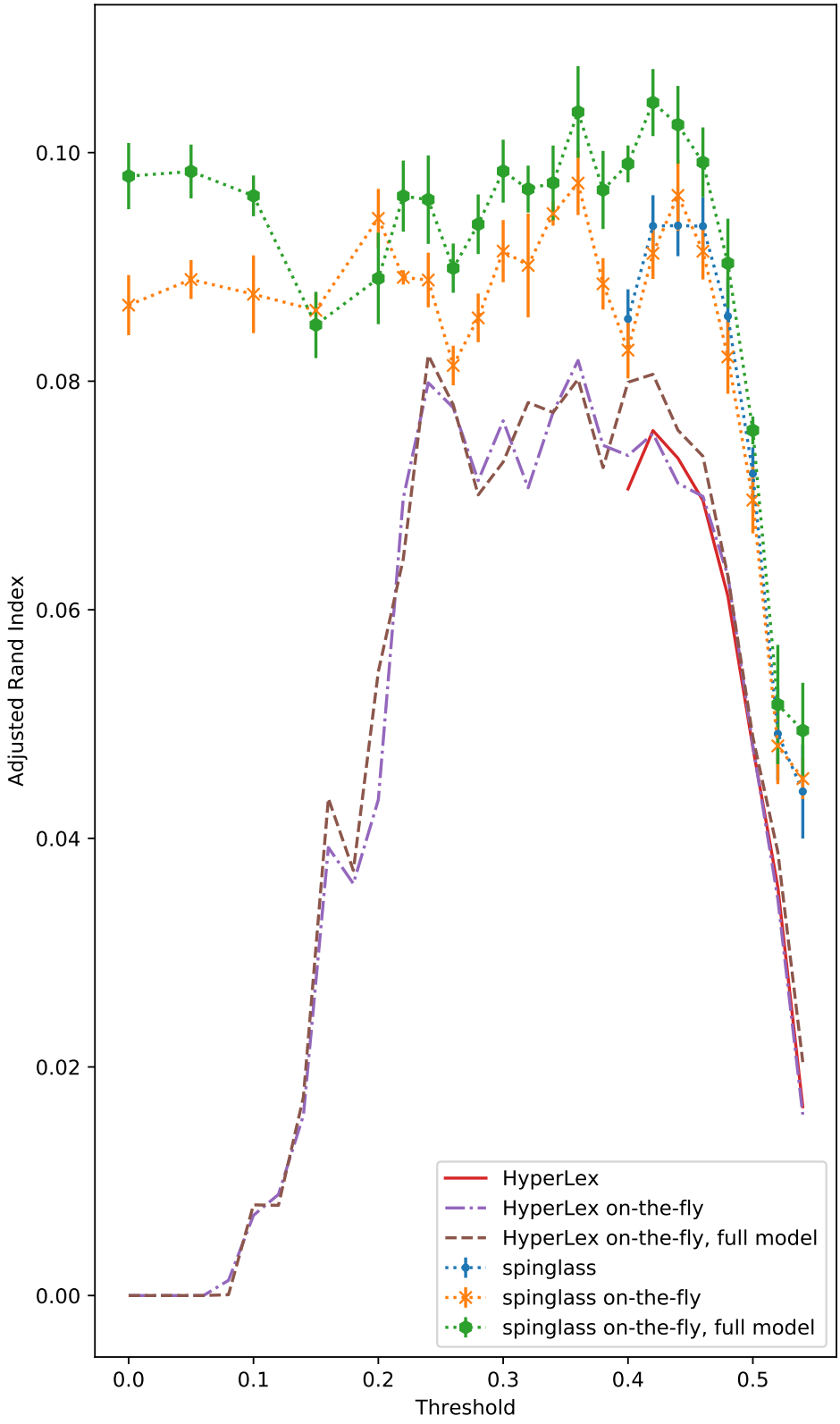


Figure 5.3: ARI scores from SemEval-2013 evaluation. Error bars show standard deviation.

Threshold	0.0	0.1	0.2	0.3	0.4	0.5
HyperLex					29	9.9
Spinglass					400	97
HyperLex, on-the-fly	868	601	397	317	300	294
Spinglass, on-the-fly	31 890	21 690	4059	1301	677	382
HyperLex, on-the-fly, full model	926	675	459	356	333	330
Spinglass, on-the-fly, full model	32 469	21 885	3916	1149	730	426

Table 5.3: Running time in seconds of WSI and WSD on the SemEval-2013 task 11 data set

when the vocabulary increases from 220 000 to 634 999 terms. This is probably because the first part of the word embedding model contains the most frequent terms. These terms are more likely to occur in contexts, and thus influence the score. The scores vary somewhat with the threshold, and sink rapidly for thresholds above 0.46. Figure 5.3 shows that for the same model size, the difference between on-the-fly and regular conversion is generally within the standard deviation.

The plot in fig. 5.3 shows that spinglass yields relatively high scores even for complete graphs made with threshold zero. However, table 5.3 shows that the spinglass running time increases rapidly with the number of edges. While spinglass is able to cluster complete graphs, this is very time-consuming. Our evaluation with threshold zero takes more than eight hours. This is because the number of edges in a complete graph is quadratic, $\binom{n}{2} = n(n-1)/2$, and probably also because the spinglass algorithm itself has polynomial complexity. The results show that dropping edges benefits not only the running time but also the ARI score for this task.

5.4.2 HyperLex

HyperLex is made to exploit the small-world structure of graph models. It is not well suited to word sense induction on complete graphs. This can be seen clearly in the results in fig. 5.3, where the score for HyperLex is very low for thresholds below 0.15. For complete or nearly complete graphs, HyperLex fails completely. HyperLex removes all neighbors of hubs or candidate senses, see section 5.1.3. In complete graphs, this means that all nodes are removed for the first sense that is found. Therefore, HyperLex will always yield only one sense for a complete graph.

For thresholds in the range 0.2 to 0.45, HyperLex does better with ARI scores between 7% and 8.5%. As with spinglass, we achieve the highest score with the largest model. The best score is around 8.23%, obtained with threshold 0.24. For the smaller model, we achieve the best score, 8.18%, for HyperLex using on-the-fly conversion and threshold 0.36. This is the same threshold as the optimal threshold for spinglass. This score is still below the scores for spinglass clustering, which suggests that spinglass is better suited to this task in terms of clustering quality. This could be because the number of clusters produced by spinglass is closer to the number of clusters in the

Target	id	ARI	Clusters	Gold clusters	Term rank
hedonism	37	0.524	6	6	54 546
queen	13	0.408	8	14	1111
bald::eagle	41	0.392	9	3	24 107
kangaroo	2	0.376	10	17	11 163
pods	32	0.332	5	5	155 631

Table 5.4: SemEval-2013 5 highest scoring target words using spinglass, $\varepsilon = 0.36$

Target	id	ARI	Clusters	Gold clusters	Term rank
fort::recovery	53	-0.021	8	5	199 594
nickelodeon	24	-0.011	9	13	10 668
far::cry	79	-0.008	5	3	39 186
aurora::borealis	76	-0.003	8	4	82 728
harry::potter	65	-0.002	6	4	12 223

Table 5.5: SemEval-2013 5 lowest scoring target words using spinglass, $\varepsilon = 0.36$

gold standard data, because the cluster members match better, or both.

However, HyperLex is much faster than spinglass. That is because HyperLex has linear complexity, while spinglass is polynomial. For some applications where speed is more important than quality, HyperLex might be preferable. This might for example be the case for web search and other on-line use. Here, the response time for the user is critical.

5.5 Error analysis

An Adjusted Rand Index of around 10% shows little correlation between our clustering results and the gold standard clusters. We therefore analyze the data to try to find the reason for the low performance. We do the error analysis on data from the evaluation on the smaller word embedding model, with 220 000 terms. The results reported in fig. 5.3 are the average ARI scores for all the 100 target words. Tables 5.4 to 5.5 lists the 5 top and bottom scoring target words, using the spinglass algorithm and $\varepsilon = 0.36$.

The highest score is achieved for the target word “hedonism”, at 52.4%. In this case, the spinglass clustering finds the same number of clusters as the gold clustering. However, for the second highest scoring word, “queen”, spinglass produces only eight clusters, only half as many as the 14 clusters in the gold data. This suggests that while a perfect result requires finding the correct number of clusters, it is possible to achieve a quite high score even without finding the correct number of clusters.

All the five lowest scoring target terms are proper nouns. Most are also quite infrequent terms, like “fort::recovery” with rank 199 549. That could

suggest that the representations of these terms are inaccurate, because of data sparsity. However, some of the highest scoring terms are also infrequent, for instance "pods" has rank 155 631. Therefore, it is more likely that the poor results for these terms are due to some properties of the clusters themselves.

The target term "fort::recovery" yields the lowest score, at -2.1%. The negative score indicates a negative correlation between our clustering and the gold data clustering. This means that search snippets belonging to the same cluster in the gold data, are more likely to belong to different clusters in our clustering results. The gold clusters or senses for the term "fort::recovery" are described as:

53.1 Fort Recovery, Ohio, a present-day village near the Ohio fort

53.2 Fort Recovery, a fort from 1793 in Ohio

53.3 Fort Recovery, Tortola, a fort from 1620 in the British Virgin Islands

53.4 Other

53.5 Fort Recovery (album), an album by Centro-Matic

The differences between these senses are quite subtle. Depending on the context, the difference between sense 1, 2 and 3 might be difficult to tell, even for humans. Thus, it is not surprising that WSI can struggle with these senses.

A good clustering result partly depends upon finding the correct number of clusters. Figures 5.4 to 5.6 shows of the distribution of the number of clusters over the 100 different terms to be disambiguated. The mean number of clusters are 7.7 for the gold standard data, 15.9 for HyperLex and 8.2 for spinglass. HyperLex is quite far from the gold standard. While the mean number of clusters for spinglass is closer to the gold standard, the distributions are different. The gold standard distribution as two maxima, at 5 and 12 clusters. Spinglass, on the other hand, has one maximum around eight clusters. This indicates that neither algorithm tends to find the correct number of clusters.

5.6 Summary

The results support our hypothesis that low similarity edges contribute little information, and should be filtered out. Dropping low similarity edges enhances the performance in terms of both speed and clustering quality. These results agree with our results in the previous chapter, where we also found it beneficial to drop some edges. We have achieved an Adjusted Rand Index score of around 10.4% on SemEval-2013. While this is not a high score, only two of the nine systems that participated in the WSI part of SemEval-2013 task 11 achieved an ARI score higher than 8% (Navigli and Vannella 2013). The two best systems, HDP-CLUSTERS-LEMMA and HDP-CLUSTERS-NOLEMMA both attained ARI scores of

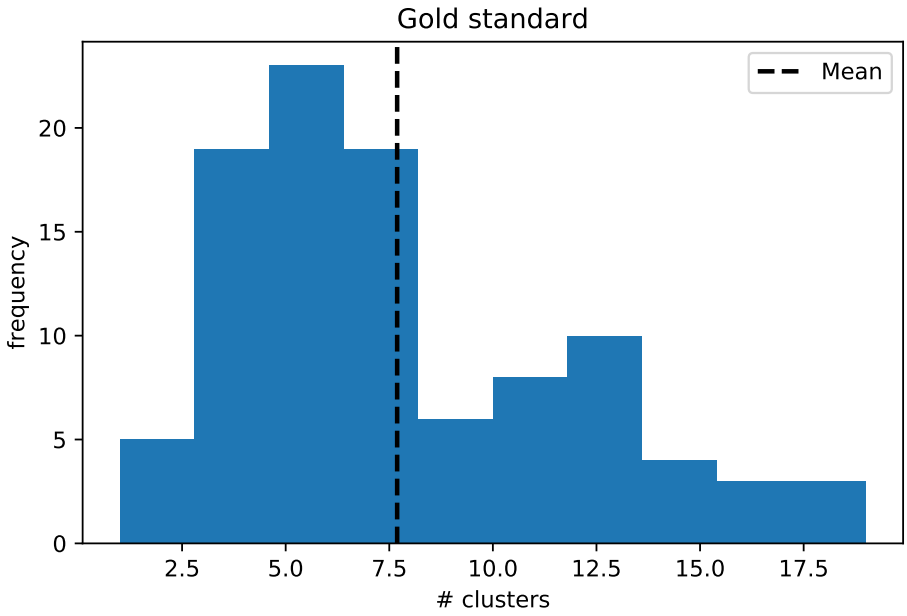


Figure 5.4: Distribution of number of clusters in gold standard data.

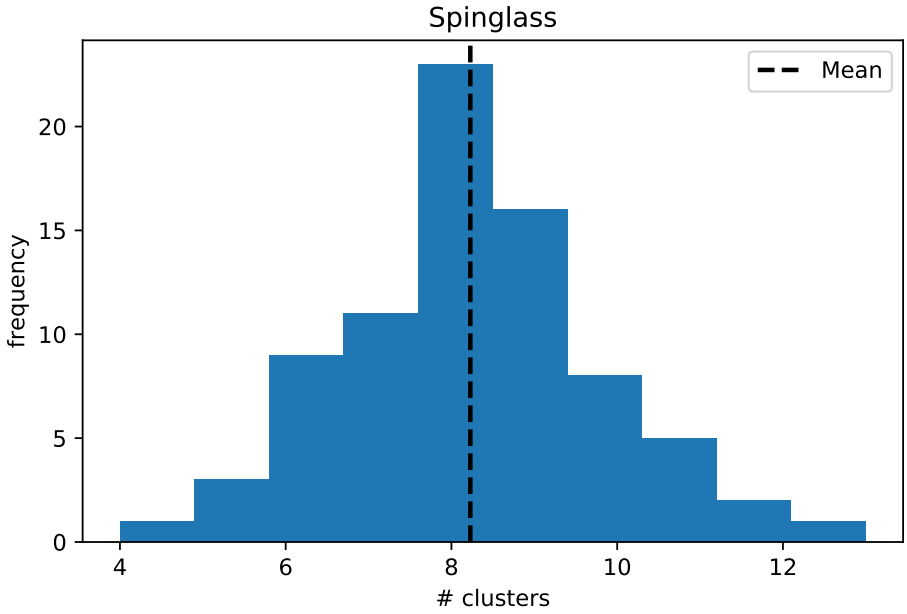


Figure 5.5: Distribution of number of clusters in spinglass clusterings, $\varepsilon = 0.36$.

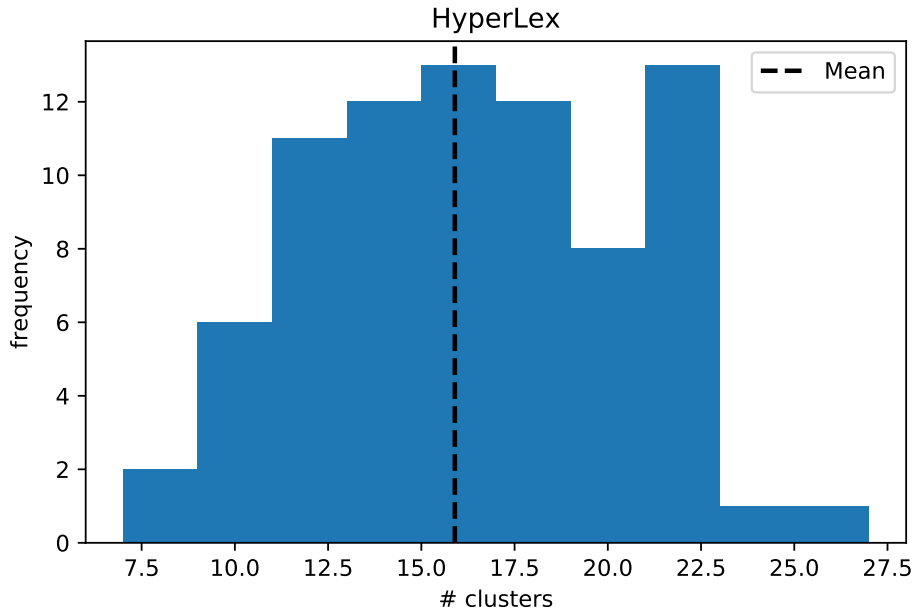


Figure 5.6: Distribution of number of clusters in HyperLex clusterings, $\varepsilon = 0.36$.

around 21%. This indicates that WSI and WSD are hard problems. Our system outperforms seven of the nine participating systems in the WSI part of this task.

The results show that both the full graph model and on-the-fly conversion can be successfully used for WSI. Converting a large word embedding model to a graph model is time-consuming and can take days for models with around 200 000 terms. Therefore, for applications where the underlying VSM model is updated frequently, or where the number of queries is relatively low, on-the-fly conversion will probably consume fewer resources. However, in an application with many queries, the time to convert the full graph model can be amortized over all the queries. Here, it will be beneficial to convert the whole model once, and extract subgraphs for each query.

For very dense graphs, say for thresholds below 0.2, doing full graph conversion is impractical. HyperLex also works poorly with such dense graphs. While it is possible to use spinglass with on-the-fly conversion, this will be time-consuming. The results make it hard to justify using such low thresholds except in applications where running time is not an issue.

For applications where clustering quality is paramount, spinglass clustering is probably preferable. However, HyperLex is a significantly faster alternative, with the trade-off of a small reduction in quality. Our results seem to imply that determining the correct number of senses is one of the largest hurdles in word sense induction. While graph clustering algorithms like Spinglass and HyperLex can be used for WSI with some success, there is still room for improvement.

As an extrinsic evaluation of our converted graph models, this experiment can be said to be a success. The evaluation shows that the converted models can be successfully used in the word sense induction and disambiguation task. The performance suggests that the conversion of the word embeddings into a graph model preserves a significant portion of the information in the model. We achieve best evaluation scores with thresholds in the range between 0.2 and 0.5. This is consistent with the results of the intrinsic evaluation in the previous chapter. The intrinsic evaluation showed that our converted models could achieve performance within 80% to 94% of the performance of the original model. In this chapter, we have shown that this translates to reasonable performance at a real-world task. In sum, the results seem to support use of converted graph models for word sense induction. In the next chapter, we do further extrinsic evaluation of our graph models. We use our graphs for hypernym discovery, and evaluate the results for different hyperparameters.

Chapter 6

Conversion evaluation by WordNet hypernym discovery

WordNet is a lexicon and *taxonomy* of the English language. A taxonomy is a hierarchical structure classifying concepts or objects. As discussed in section 2.3.2, WordNet defines many different relations between *synsets*. One of the most important relations, which defines the hierarchical structure of WordNet, is the hypernym/hyponym relation. A hypernym is a more general, superordinate concept, such as "vehicle", and a hyponym is a more specific subconcept, such as "car". The hypernym/hyponym relation is also referred to as the *is-a* relation.

Here, we do an explorative study to see if we can use centrality in local graph models as an indicator of hypernymy. We do this as further extrinsic evaluation of our converted graph model. We evaluate the performance of the graph model in an application for hypernym discovery. If we find a correspondence between centrality and hypernymy, it might be possible to use our models for taxonomy extraction or hypernym discovery. *Taxonomy extraction* entails automated creation of a taxonomy from language resources like a corpus. *Hypernym discovery* is a subtask of taxonomy extraction that focuses only on extracting hypernym/hyponym relations. Hypernym discovery systems use various techniques and features, for instance substring matches or syntactic Hearst patterns (Bordea, Lefever and Buitelaar 2016). We use WordNet as our source of hypernym/hyponym relations.

In the next section, we discuss previous work on hypernym discovery. In section 6.2, we discuss our implementation of hypernym discovery using graph centrality. Then, in section 6.3, we discuss our evaluation results. In section 6.4, we do an error analysis on our results. Finally, in section 6.5 we summarize our results in this chapter.

6.1 Previous work

Both supervised and unsupervised learning are used for hypernym discovery. *Pattern based* methods and *distributional* or *distributional semantic* methods are two major approaches to this task (C. Wang, He and Zhou 2017). Distributional semantic methods employ distributional semantic information, like word embeddings, in various ways. The term *distributional* method might arguably be a slight misnomer. This is because pattern based methods might also be said to utilize distributional information, in that a pattern specifies the distribution of words in a text.

One of the most common pattern based methods is lexico-syntactic patterns, also known as *Hearst patterns* (Hearst 1992). Hearst patterns are unsupervised, and have long been used for hypernym discovery. They are simple patterns like "X such as Y". "Vehicles such as cars…" is an example of a text fragment matching this pattern. From this, we can infer that "vehicle" is a hypernym of "car". Hearst patterns have high precision, but low recall. High precision is achieved because texts will rarely contain expression matching a pattern, where the hypernym/hyponym relation does not hold. The recall is low because there will be many hypernym/hyponym pairs that do not occur together in a text. However, Hearst patterns can outperform unsupervised distributional methods for some tasks, like direction predicting (Roller, Kiela and Nickel 2018). This is because patterns may capture more detailed information about the context, in this case the direction of the *is-a* relationship.

Substring matching is a simple, rule-based method utilizing compositionality of terms. With substring matching, a term is assumed to be a hypernym of a candidate term if it is contained in the candidate. For instance, substring matching will infer "fish" to be a hypernym of "flying fish". Despite its simplicity, this method can be quite effective (Bordea, Buitelaar et al. 2015). However, it will also yield some incorrect results. For example, substring matching will incorrectly infer "apple" to be a hypernym of "pineapple".

Word embedding methods are distributional machine learning methods for hypernym discovery. They are based on word embeddings and word embedding arithmetic or projections. Unlike pattern-based methods, these methods utilize learned vector representations of words. The word embeddings are often pre-trained. Word embeddings are often used in supervised systems. The system learn a function that classifies whether or not a hypernym and candidate word has a hypernym/hyponym relation. This function is learned from data consisting of hyponym/hyponym pairs, as well as pairs of nonrelated words as negative examples. This is often done by learning a *projection* of terms, so that the nearest neighbor of the projected term is its hypernym. Different projection matrices can be used to represent different kinds of hypernymy (Bernier-Colborne and Barriere 2018).

There are also hybrid systems which combine distributional semantic and pattern methods. These can for example create separate lists of hypernym candidates with each method, and then merge the lists, giving higher scores to candidates that occur in both lists. CRIM (Bernier-

Colborne and Barriere 2018) is a recent example of a hybrid system. It achieved top score in three of the five subtasks of hypernym discovery in SemEval-2018 Task 9 (Camacho-Collados et al. 2018).

6.2 Implementation

We use the *Natural Language Toolkit* (Bird and Loper 2004), also known as NLTK, to work with WordNet. We find all suitable hypernyms in WordNet. These hypernyms should:

- Be mid-frequent, that is have rank in the range $[1000, 100\,000]$ in the word embedding model
- Have at least five hyponyms in the word embedding model

WordNet contains more than 1000 hypernyms matching these criteria. For each suitable hypernym, we create a neighborhood graph containing the hypernym and its hyponyms from WordNet. We include only terms that exist in the vocabulary of the word embedding model.

6.2.1 Graph construction

We create each graph using *on-the-fly* conversion. The conversion is done using the threshold method, as described in section 3.2.1. This method adds an edge between two nodes only if their similarity is above the threshold ε. This limits the number of edges in the graph. We evaluate the correspondence between centrality and hypernymy for different thresholds.

The terms to include are from WordNet, while the edges are generated from the word embedding model by the conversion procedure. Figure 6.1 shows a plot of a graph constructed for the hypernym “lamp”.

Some centrality measures consider the edge weights when calculating the centrality. Edge weights commonly indicate some measure of *distance* between two nodes, with lower weights meaning smaller distance. However, in our graphs high weights indicate high similarity, with lower weights meaning lower similarity and thus larger distance. Therefore, we invert the weights in our graph before calculating centrality. We do this by setting the weights to $w = 1/w$.

WordNet groups lemmas into synsets, and we only include one lemma from each synset in our graphs. Since WordNet orders the lemmas by frequency, we use the first lemma from each synset that is also in the word embedding model. Using the most frequent lemmas should be beneficial, since they should also have higher quality word embeddings.

6.2.2 Hypernym proposal

When the graph has been constructed, we run a graph centrality measure to find the most central node in the graph. The centrality measure returns

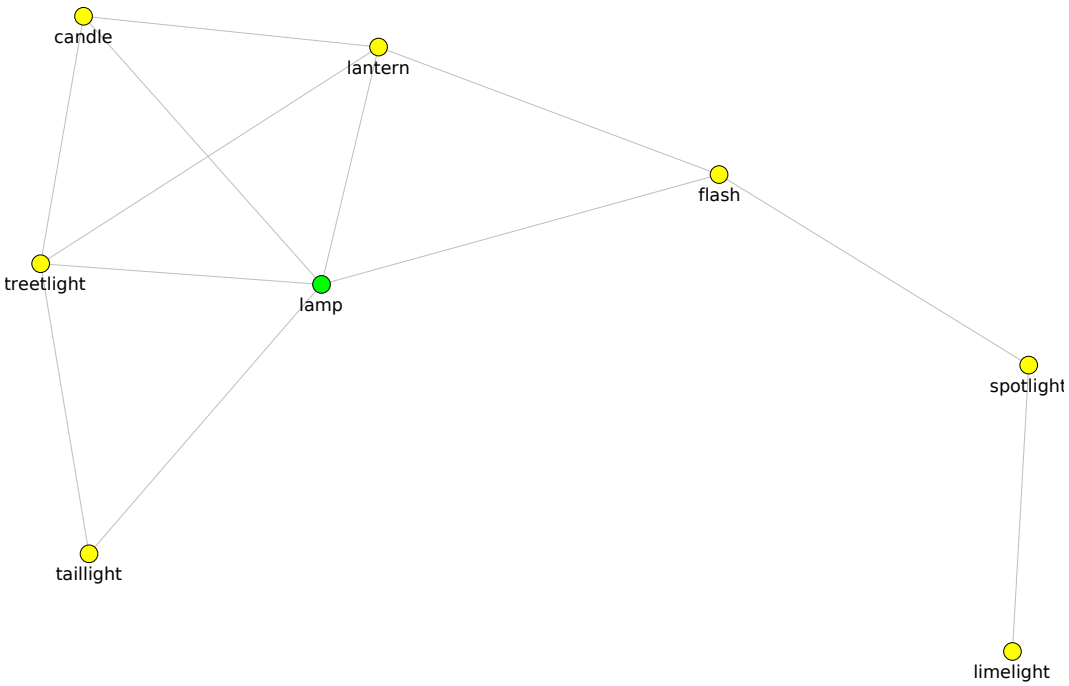


Figure 6.1: Graph with the hypernym “lamp” and its WordNet hyponyms

the centrality score of each node in the graph. We use this score to sort the nodes by their centrality, and pick the top scoring one. This most central node is proposed as a prediction of the hypernym. For some graphs, two or more nodes might be tied for the highest centrality score. Therefore, we need some way of breaking ties. This is discussed in section 6.2.6.

6.2.3 Centrality

We are looking for correspondence between hypernyms and central nodes in our WordNet graphs. Centrality is a measure of the importance of a graph node. This can be defined in different ways, for instance central nodes can be nodes that are important for the overall connectivity of the graph, or nodes with many neighbors. Thus, there are several different measures of node centrality in a graph. We evaluate three different measures:

Betweenness measures the number of shortest paths a node lies on. Betweenness is high for nodes that are on the shortest paths between many different pairs of nodes. The betweenness measure can use edge weights when calculating shortest paths. We therefore supply the edge weights as an argument to the algorithm.

PageRank measures the importance of a node by the number of its neighbors, and their importance. It was originally developed by

					Baseline accuracy	
Algorithm	NLPL id	Corpus	Vocabulary	Hypernyms	Random	Frequency
Skip-gram	5	Wikipedia	273 992	1248	0.115	0.288
Skip-gram	11	Gigaword	261 794	1078	0.116	0.242

Table 6.1: Models used in WordNet exploration

Google for web search.[1] PageRank also considers edge weights when calculating centrality.

Degree centrality is the number of edges adjacent to the node. The degree centrality is high for nodes with many neighbors. This centrality measure does not take edge weights into account.

6.2.4 Models

We evaluate the accuracy of using centrality to predict hypernymy in graph models produced by on-the-fly conversion from word embedding models. As in section 4.2 we use models from the NLPL word embeddings repository. We run the evaluation with two different word embedding models, produced from two different corpora. The models are built on the Gigaword corpus and Wikipedia, and are listed in table 6.1. Both models are word2vec skip-gram models built with gensim. We use two different models to gauge the impact of the corpus on the accuracy.

6.2.5 Evaluation

For each graph, we check whether the proposed hypernym is the actual hypernym. Each graph contains exactly one hypernym. We assign a binary score of zero or one to each graph. If the hypernym is identical to the proposed hypernym, the score is one, otherwise zero. The results are then averaged over all the suitable WordNet hypernyms, yielding a total score between zero and one. This score measures the overall accuracy of our predictions. We run this evaluation procedure with different threshold values ranging from zero to one, with each of the different centrality measures.

6.2.6 Centrality ties

We can expect that two or more nodes will be tied for top centrality in some cases. Both the betweenness and degree centralities yield integer values. Betweenness centrality measures the number of shortest paths each node is part of. This is limited upwards by the maximum number of combinations of two nodes in the graph, which is $\binom{n}{2} = n(n-1)/2$. Degree centrality measures the degree of, that is the number of edges adjacent to, each node.

[1] We use Personalized PageRank, because of a bug in igraph's PageRank implementation that causes a segmentation fault.

This is limited by the number of nodes in the graph. In either case, each node is assigned a centrality score from a quite limited range, so collisions are likely to occur. In some cases, this can lead to a tie for the highest centrality score. Therefore, we need some way to break ties in order to choose our proposed hypernym.

Breaking ties arbitrarily

The simplest approach is to select an arbitrary node when there are ties. When the nodes have been sorted by centrality, we can simply select the first node in the list, without checking for ties. This requires no extra processing, and can be sufficient when speed is more important than accuracy. However, depending on the implementation the results might be nondeterministic.

Breaking ties by term frequency

The terms in our word embedding models are ordered by decreasing frequency. We hypothesize that this information can be useful for proposing hypernyms. Figure 6.1 shows the hypernym "lamp" and its WordNet hyponyms. We can expect many of the hyponyms, like "spotlight" and "limelight" to be less frequent than "lamp". Therefore, we attempt to improve our hypernym proposals by selecting the most frequent term in the event of a tie. This is implemented by first sorting the nodes in the graph by frequency, and then by centrality. When implemented with a *stable* sorting algorithm, this ensures that nodes with the same centrality score is ordered by frequency.

6.2.7 Baselines

An accuracy score in itself does not tell much unless we can compare it with something. Therefore, we have computed two baseline scores to use as a basis of comparison for our results. These baselines were calculated from the hypernym and hyponym terms, without any use of graphs.

Random baseline

The random baseline accuracy was computed as the expected score for each graph. Each graph consists of one hypernym and all of its WordNet hypernyms. If a graph contains N terms in all, the chance of randomly picking the right term as hypernym is $1/N$. The overall baseline score is the mean of the baseline scores for each graph.

The random baseline accuracy is 0.116 for the Gigaword model, and 0.115 for the Wikipedia model. This means that our hypernyms have around eight hyponyms on average. Table 6.1 lists the baseline scores for our models.

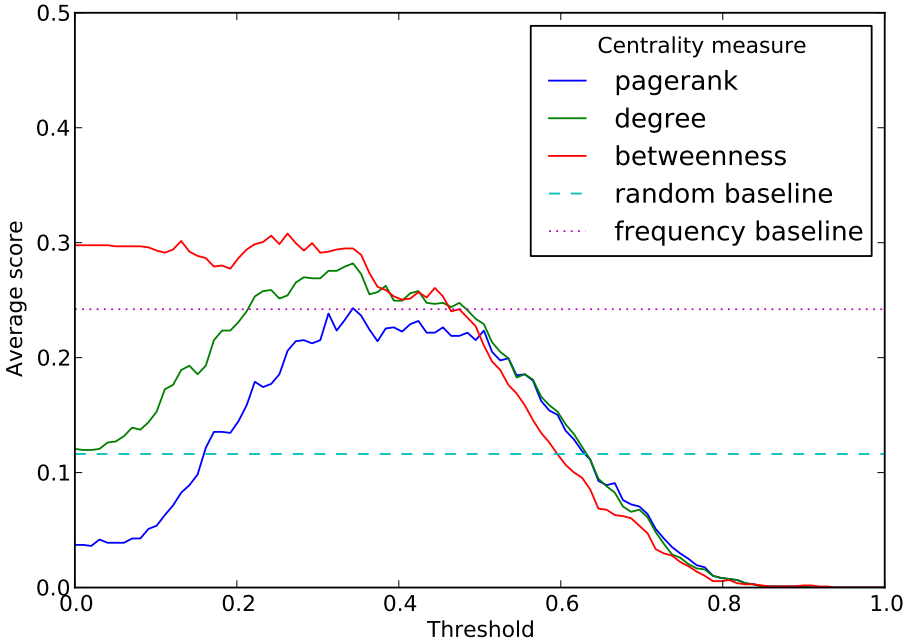


Figure 6.2: Accuracy of hypernym predictions with Gigaword-based models. Ties are broken arbitrarily.

Frequency baseline

In many cases, a hypernym is more common than its hyponyms. Figure 6.1 shows the hypernym "lamp" and its WordNet hyponyms. Here, we can expect many of the terms, like "spotlight" and "limelight" to be less frequent than "lamp". Thus, if we assume that the most frequent term is the hypernym, we should get a reasonable baseline. Our word embedding models are ordered by frequency rank. We use this ordering to sort the terms in our hypernym/hyponym sets by frequency.

We calculate the frequency baseline by selecting the most frequent term in each set of hypernym/hyponyms. This proposed hypernym is then evaluated with the usual scoring function, as described in section 6.2.5.

The frequency baseline is 0.242 for the Gigaword model, and 0.288 for the Wikipedia model. In other words, the hypernym is the most frequent term in around 25% of the cases. This is significantly better than the random baseline.

6.3 Results

The Wikipedia and Gigaword models contain respectively 1248 and 1078 hypernyms matching our criteria. The accuracy of our hypernym predictions varies greatly with the threshold and the centrality measure we use.

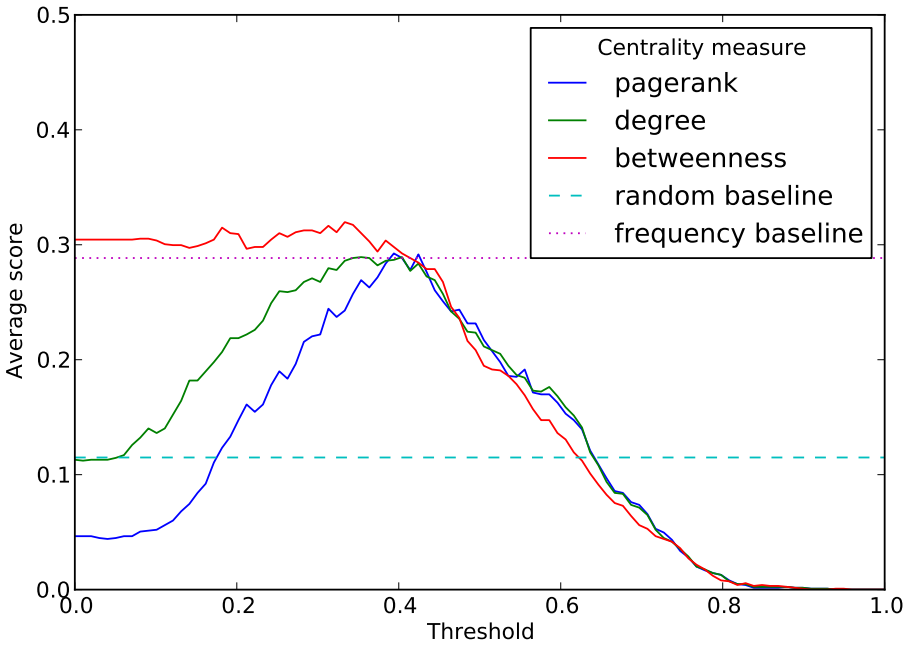


Figure 6.3: Accuracy of hypernym predictions with Wikipedia-based models. Ties are broken arbitrarily.

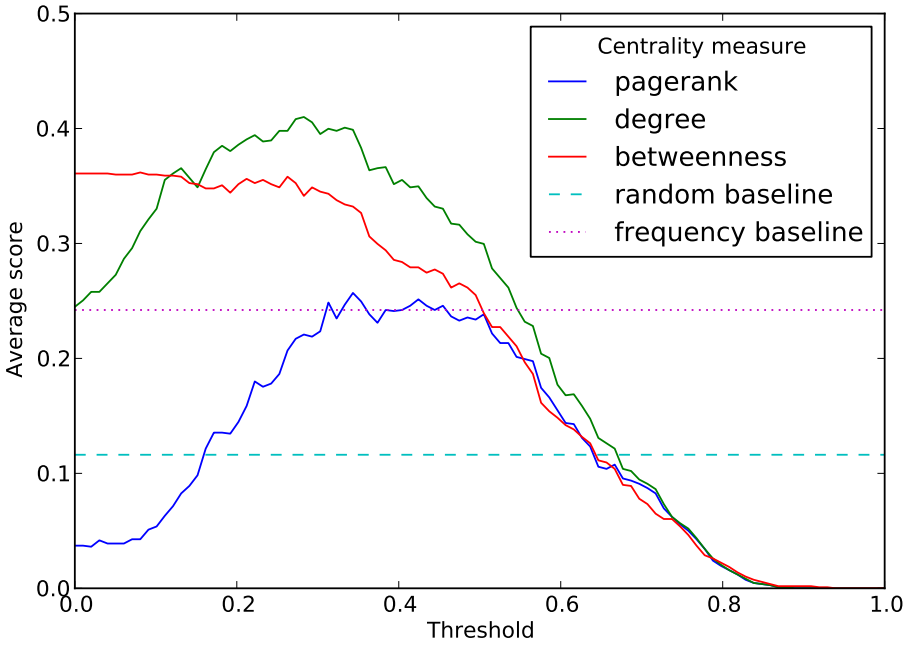


Figure 6.4: Accuracy of frequency assisted hypernym predictions with Gigaword-based models

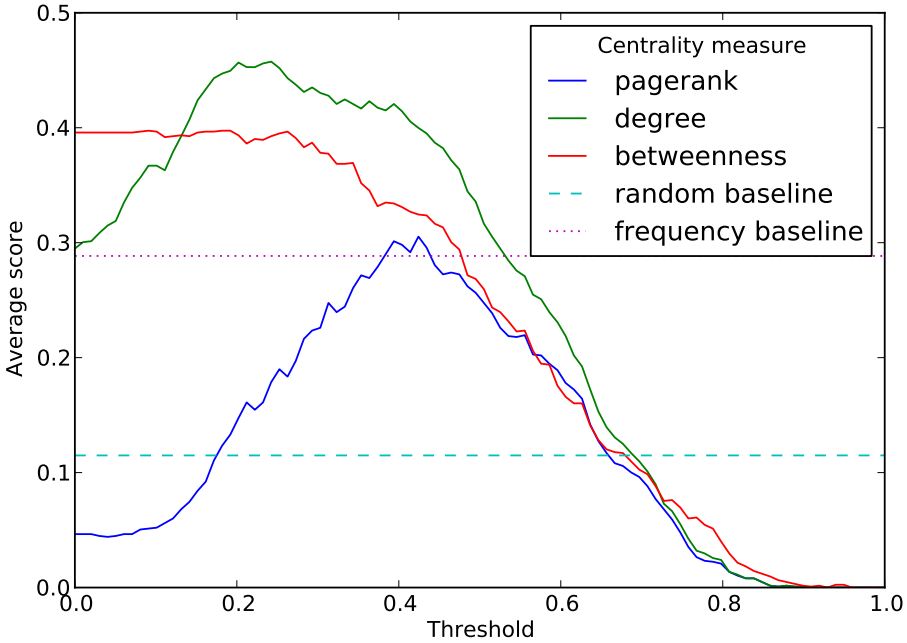


Figure 6.5: Accuracy of frequency assisted hypernym predictions with Wikipedia-based models

6.3.1 Ties broken arbitrarily

Table 6.2 lists the best thresholds and accuracy scores for the various centrality measures, while fig. 6.2 and fig. 6.3 plots the accuracy scores for different thresholds. For all three centrality measures, the best accuracy is achieved with thresholds in the interval 0.2-0.5. This is consistent with the results for word sense induction in section 5.4. There, the optimal threshold values also lie in the range from 0.2 to 0.5.

The plots in fig. 6.2 and fig. 6.3 show that the betweenness measure achieves a high accuracy even with low thresholds. This suggests that the betweenness algorithm benefits appreciably from using edge weights. With the betweenness measure, there is only small variation in the accuracy for thresholds in the range from 0.0 to around 0.3.

Gigaword model

Figure 6.2 shows the accuracy of the hypernym predictions for graphs based on the Gigaword model. Betweenness is the best overall centrality measure. The best score of 0.308 is obtained with the threshold 0.263.

The degree centrality is second best in this configuration of hyper-parameters, and beats the frequency baseline for substantial range of thresholds, from around 0.2 to 0.5. PageRank yields the lowest accuracy for this configuration, lower than the frequency baseline. This suggests that the frequency baseline is quite strong.

Wikipedia model

Figure 6.3 shows the accuracy of our predicted hypernyms for graphs based on the Wikipedia model. Here, the betweenness centrality is also best overall. The best accuracy, 0.320, is obtained with a threshold of 0.333. In addition to this global optimum, the betweenness score has several local optima, for instance around threshold 0.18, where the accuracy is around 0.31.

The degree centrality is second best for a wide range of thresholds for this model as well. However, PageRank has a peak for threshold 0.394 which beats the degree centrality by a small margin. For this model, only betweenness manages to beat the frequency baseline by a significant margin. PageRank and degree centrality only matches the frequency baseline.

6.3.2 Ties broken by term frequency

Table 6.3 lists the best thresholds and accuracy scores for the different centrality measures when ties are broken by frequency. These scores are considerably higher than the results above, where ties are broken arbitrarily. Both betweenness and degree centrality beat the frequency baseline by a large margin. This confirms our hypothesis that term frequency is useful in predicting hypernymy.

Gigaword model

Figure 6.4 shows the accuracy of the frequency assisted hypernym predictions for the Gigaword model. While the optimum threshold varies between the different centrality measures, the peaks mostly lie in the range between 0.2 and 0.5. The only exception is the betweenness centrality, which has its maximum score for thresholds around 0.08. However, this optimum is lower than the one for the degree centrality. For this setup, the degree centrality outperforms betweenness by a clear margin. The best score of 0.410 is obtained with a threshold of 0.283. It is clear that degree centrality benefits more from frequency information than does betweenness. The reason for this is probably that the degree centrality has more ties.

Table 6.3 shows that the degree centrality results in ties for 406 graphs, while the number is only 201 for betweenness. Since we use 1078 hypernyms from the Gigaword model, this means that around 38% of the graphs result in ties with the degree centrality. As discussed in section 6.2.6, the degree centrality has a smaller range of output values than betweenness. Degree centrality is limited upwards by the number of nodes in the graph. Since there are fewer values to choose from, more nodes are likely to be assigned the same value, and there will probably be more nodes in a tie than with betweenness. In all, we can expect the degree centrality to yield more ties involving more terms than betweenness. This means that there is more use for the frequency information, and we have more nodes in a tie to "choose" from.

Model	Centrality	A	Best ε	# FC	A FC	A ex. FC	# ties	A ties
Gigaword	Random baseline	0.116	–	–	–	–	–	–
Gigaword	Frequency baseline	0.242	–	–	–	–	–	–
Gigaword	PageRank	0.243	0.343	31	0.097	0.247	48	0.375
Gigaword	Degree	0.282	0.343	31	0.097	0.287	373	0.351
Gigaword	Betweenness	0.308	0.263	75	0.080	0.325	195	0.221
Wikipedia	Random baseline	0.115	–	–	–	–	–	–
Wikipedia	Frequency baseline	0.288	–	–	–	–	–	–
Wikipedia	PageRank	0.292	0.394	50	0.180	0.297	48	0.417
Wikipedia	Degree	0.289	0.354	74	0.149	0.298	425	0.278
Wikipedia	Betweenness	**0.320**	0.333	86	0.140	0.333	242	0.194

Legend: FC: fully connected, A: accuracy, ex.: excluding

Table 6.2: Accuracy scores with ties broken arbitrarily

Wikipedia model

Figure 6.5 plots the accuracy scores of our hypernym predictions for the Wikipedia model, when ties are broken by term frequency. Again, the optimum threshold varies between the different centrality measures, and the peaks mostly lie in the range between 0.2 and 0.5. The exception is again the betweenness centrality, which has its maximum score for thresholds around 0.09. Like for the Gigaword model, the degree centrality outperforms the two other centrality measures by a substantial margin. The best accuracy is 0.458, and is obtained with a threshold of 0.242 using the degree centrality. This optimal threshold is somewhat smaller than for the Gigaword-based model. Still, it is within the range from 0.2 to 0.5 that tends to produce the best results. This lower optimum might be an arbitrary effect, or it can be due to some interaction between the centrality scores and the frequency data.

The accuracies for the Gigaword-based and Wikipedia-based models are roughly the same. The difference in accuracy can probably be explained by the different vocabularies of the two models. Yet, even the best accuracy score is only moderate. We therefore do an error analysis to attempt to find the reason for this low accuracy.

6.4 Error analysis

We have produced only moderately accurate predictions of hypernyms. The results above show that the hypernym is the most central word in less than one third of the graphs we have used for evaluation. We therefore examine the data in an attempt to find reasons for this. All the data and plots used here are for the best result with the Wikipedia model, ties broken arbitrarily, threshold 0.333 and the betweenness centrality measure, unless otherwise specified.

Table 6.4 lists the evaluation data for the first 40 of our selected WordNet hypernyms. The first 10-15 lines contain terms from the top

Model	Centrality	A	Best ε	# FC	A FC	A ex. FC	# ties	A ties
Gigaword	Random baseline	0.116	–	–	–	–	–	–
Gigaword	Frequency baseline	0.242	–	–	–	–	–	–
Gigaword	PageRank	0.257	0.343	31	0.097	0.262	48	0.688
Gigaword	Degree	0.410	0.283	65	0.554	0.401	406	0.672
Gigaword	Betweenness	0.362	0.081	481	0.407	0.325	201	0.567
Wikipedia	Random baseline	0.115	–	–	–	–	–	–
Wikipedia	Frequency baseline	0.288	–	–	–	–	–	–
Wikipedia	PageRank	0.305	0.424	37	0.135	0.310	73	0.685
Wikipedia	Degree	**0.458**	0.242	186	0.570	0.438	602	0.669
Wikipedia	Betweenness	0.397	0.091	646	0.440	0.352	320	0.541

Legend: FC: fully connected, A: accuracy, ex.: excluding

Table 6.3: Accuracy scores with ties broken by term frequency

levels of the WordNet hierarchy, and are probably not quite representative of the rest of the data. The term "substance" occurs twice, representing two different synsets with different meanings. Of the actual hypernyms listed in the table, around half are identical with the proposed hypernyms, that is centers. This is better than the 32% overall accuracy. The table shows that there can be a substantial difference between the centrality of the proposed and actual hypernyms. Below, we examine the graphs of selected hypernyms in greater detail.

There appears to be three major error sources that contribute to our low accuracy scores, which we detail below. For comparison, fig. 6.6 illustrates a case where the hypernym is the most central term.

6.4.1 Fully connected graphs

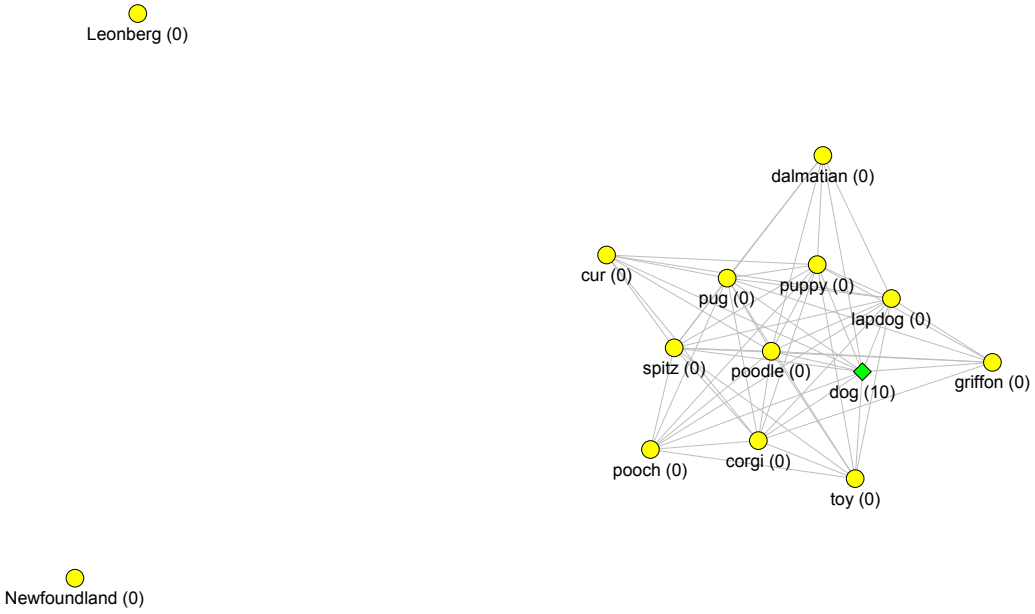
Even though some centrality measures take edge weight into account, fully connected graphs make it harder to find or even define the most central node. This is especially true when the edge weights are similar. In many cases the centrality measure yields zero for all nodes.

Table 6.2 lists the number of fully connected graphs for the different optimal thresholds. As expected, lower thresholds yield more edges, and therefore more fully connected graphs. (These numbers are not comparable across different models, that is Wikipedia and Gigaword.) Table 6.2 shows that of the 1248 graphs based on the Wikipedia model, 86 are complete using the optimal threshold for the betweenness measure. For these fully connected graphs, the accuracy is 14.0%, which is less than half the total accuracy of 32.0%. Table 6.3 also lists the scores we obtain when excluding fully connected graphs. For the betweenness centrality, the accuracy increases from 32.0% to 33.3% when complete graphs are excluded. Thus, even though the betweenness measure can in theory find the center of complete graphs based on the weights, this fails in many cases. This might be because the weights are too similar.

For instance, fig. 6.7 shows the graph for the term "hawk" and its hyponyms. This graph is fully connected, and all the nodes get a

		Centrality		
Hypernym	Center	Hypernym	Center	Difference
abstraction	set	0.0	0.0	0.0
object	object	66.0	66.0	0.0
whole	assembly	0.0	0.0	0.0
organism	organism	85.0	85.0	0.0
substance	substance	11.0	11.0	0.0
substance	humectant	3.0	66.0	63.0
matter	substance	0.0	12.0	12.0
artifact	paving	0.0	209.0	209.0
cognition	perception	16.0	20.0	4.0
attribute	trait	0.0	29.0	29.0
feeling	emotion	47.0	71.0	24.0
shape	shape	10.0	10.0	0.0
relation	causality	2.0	36.0	34.0
communication	message	0.0	6.0	6.0
phenomenon	phenomenon	6.0	6.0	0.0
kindness	kindness	1.0	1.0	0.0
accomplishment	accomplishment	15.0	15.0	0.0
departure	departure	1.0	1.0	0.0
discovery	discovery	8.0	8.0	0.0
propulsion	throw	10.0	21.0	11.0
recovery	rescue	0.0	2.0	2.0
touch	touch	20.0	20.0	0.0
arrival	anchorage	0.0	2.0	2.0
entrance	intrusion	0.0	2.0	2.0
withdrawal	retreat	0.0	2.0	2.0
failure	failure	3.0	3.0	0.0
mistake	mistake	11.0	11.0	0.0
blunder	bobble	1.0	1.0	0.0
acquisition	inheritance	1.0	1.0	0.0
seizure	seizure	1.0	1.0	0.0
rescue	rescue	1.0	1.0	0.0
liberation	emancipation	0.0	1.0	1.0
throw	throw	13.0	13.0	0.0
pitch	pitch	7.0	7.0	0.0
push	push	2.0	2.0	0.0
pull	pull	1.0	1.0	0.0
shooting	shoot	0.0	5.0	5.0
connection	interconnection	10.0	15.0	5.0
fastening	bonding	4.0	6.0	2.0
determination	validation	0.0	3.0	3.0

Table 6.4: Excerpt of WordNet betweenness centrality evaluation data for the Wikipedia model, breaking ties arbitrarily.



Green diamond: hypernym, which is also most central

Figure 6.6: The hypernym "dog" and its WordNet hyponyms

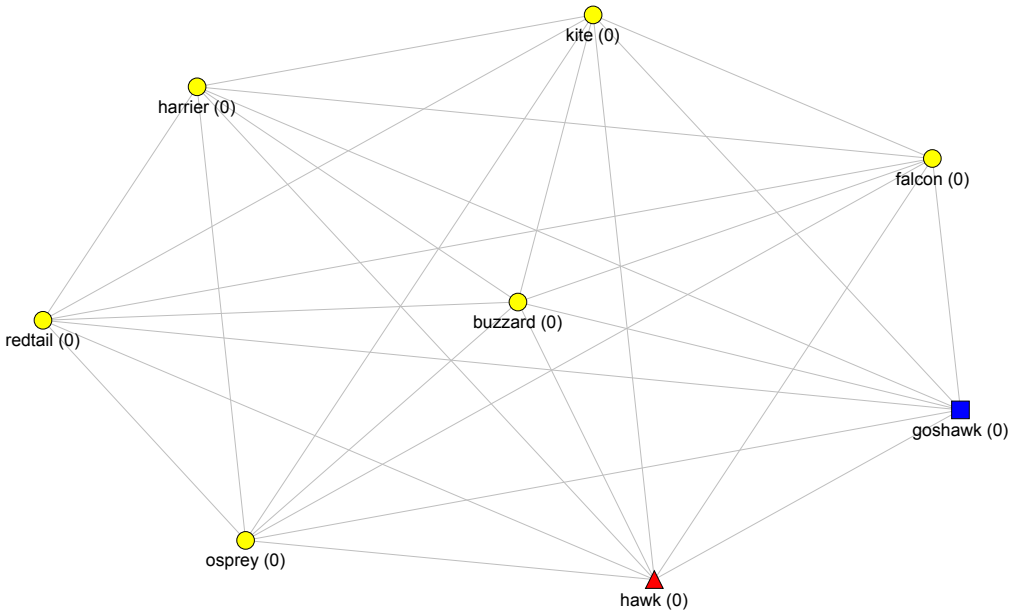
betweenness score of zero. The betweenness scores are given in parenthesis for each node in the plot. Figure 6.8 is another example of a fully connected graph. Here, all nodes also get a betweenness of zero. In both cases, our algorithm selects an arbitrary node as the proposed hypernym. "Hawk" and "protein" are both concrete terms, but abstract terms can also yield fully connected graphs. "Dislike" is an example of this, shown in fig. 6.9.

When we break ties by choosing an arbitrary node, we will sometimes choose the correct hypernym by happy accident. This will happen with probability $p = 1/N$, where N is number of nodes in the subgraph.

6.4.2 Ties

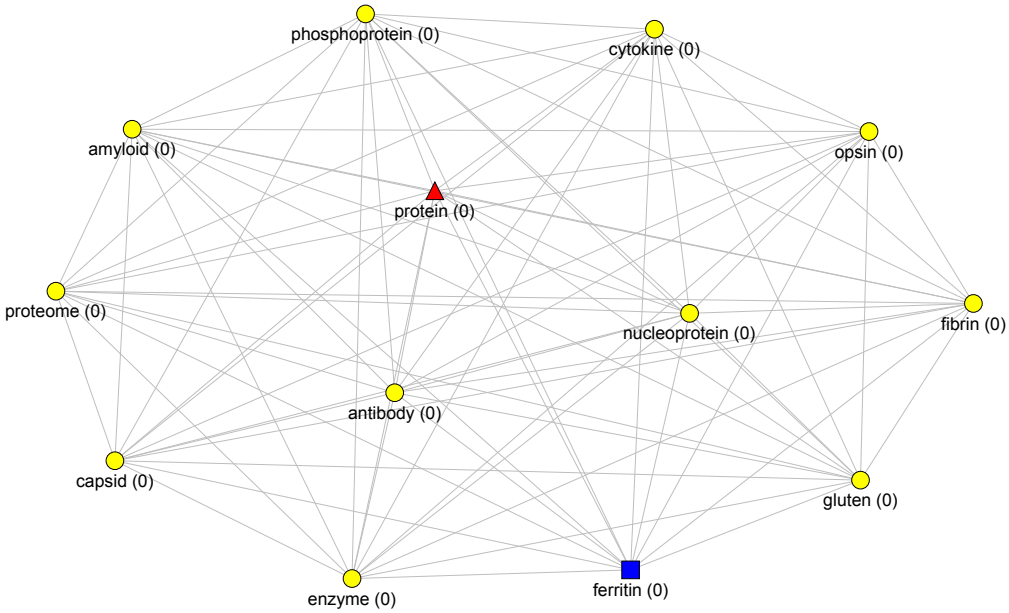
In addition to complete graphs, which often result in ties, other graphs can also have ties. Then it is also impossible to determine a single most central node. However, it might be possible to consider all the most central nodes as candidate hypernyms.

Table 6.2 shows that the number of ties varies with the threshold and centrality measure. Using the betweenness centrality, 242 of the 1248 graphs based on the Wikipedia model have two or more nodes that are tied for the most central node. This constitutes 19% of the graphs, while for Gigaword models the number of ties is slightly lower at 18%. Table 6.2 also lists the accuracy score for graphs where there are ties. The accuracy for the betweenness centrality with the Wikipedia model is 0.194. This is still



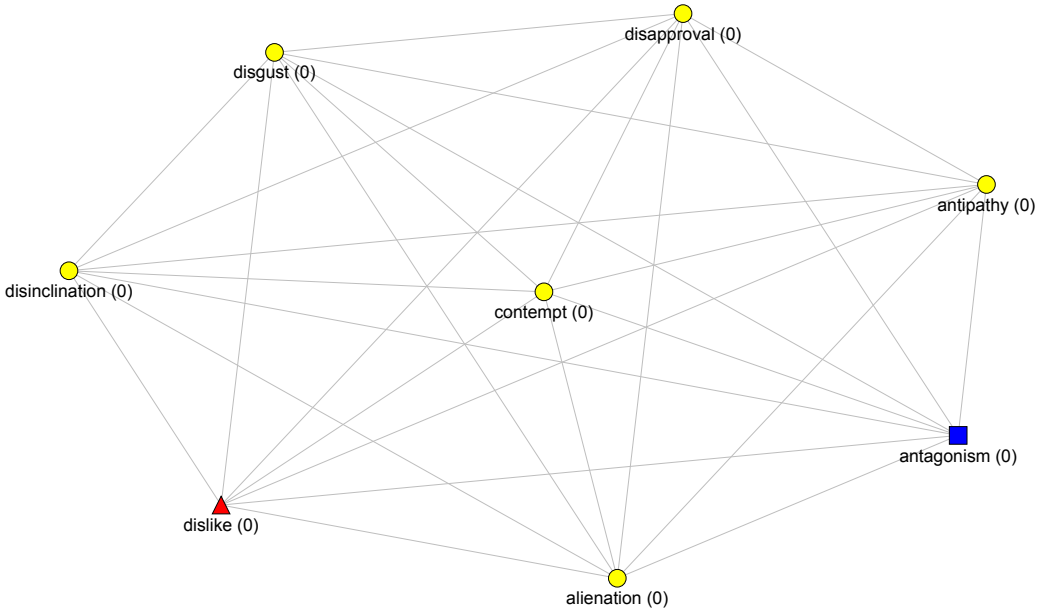
Red triangle: hypernym, blue square: highest betweenness, centrality score in parenthesis

Figure 6.7: The hypernym “hawk” and its WordNet hyponyms



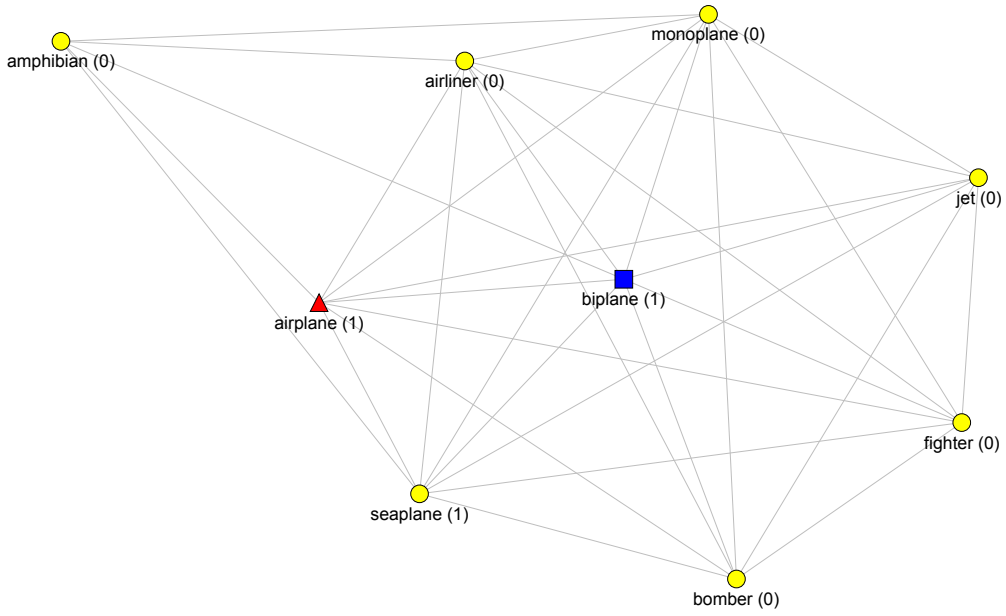
Red triangle: hypernym, blue square: highest betweenness, centrality score in parenthesis

Figure 6.8: The hypernym “protein” and its WordNet hyponyms



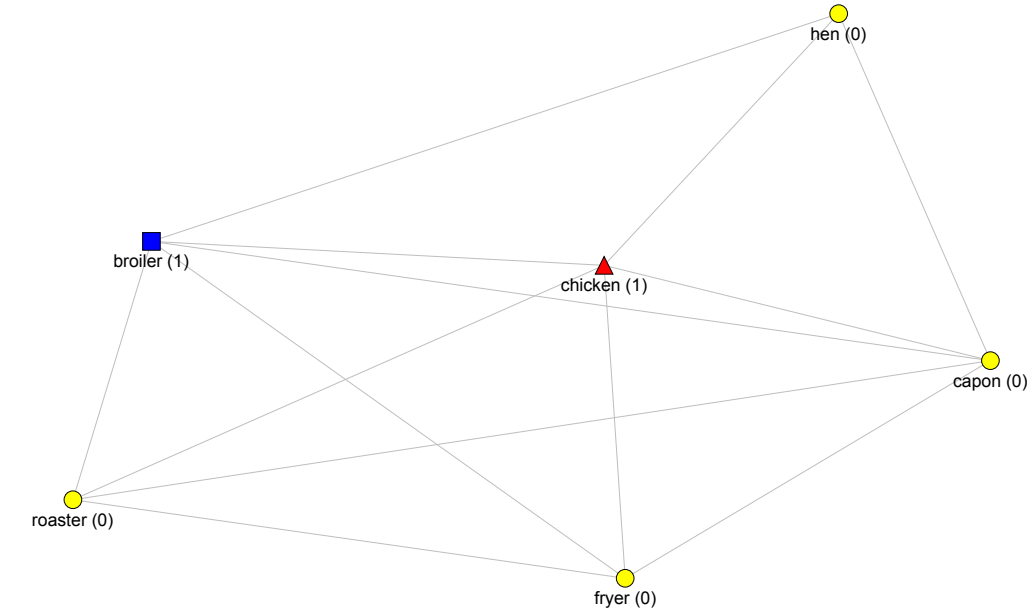
Red triangle: hypernym, blue square: highest betweenness, centrality score in parenthesis

Figure 6.9: The hypernym “dislike” and its WordNet hyponyms



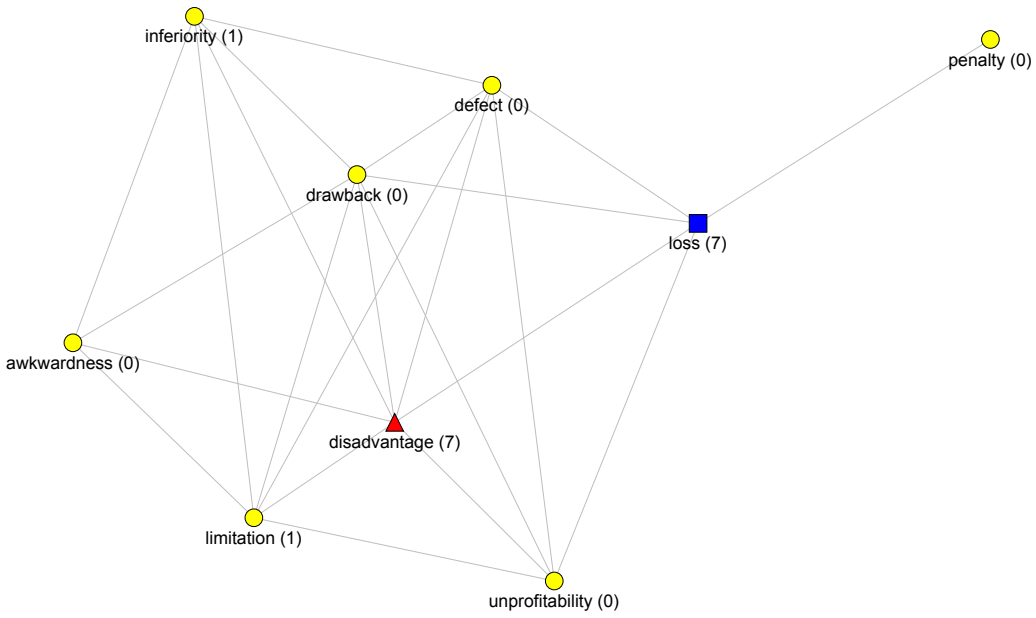
Red triangle: hypernym, blue square: highest betweenness

Figure 6.10: The hypernym “airplane” and its WordNet hyponyms



Red triangle: hypernym, blue square: highest betweenness

Figure 6.11: The hypernym “chicken” and its WordNet hyponyms



Red triangle: hypernym, blue square: highest betweenness

Figure 6.12: The hypernym “disadvantage” and its WordNet hyponyms

better than the random baseline.

We could increase hypernym recall, at the cost of precision, by including all the terms included in a tie as proposed hypernyms. Proposing multiple hypernyms makes most sense when only a few terms are involved in the tie. Giving more than 5 or 10 candidates is probably not very helpful. In the extreme case of fully connected graphs, all the nodes often have the same centrality score of zero. In this case, proposing all the terms as candidate hypernyms does not give any information at all. Therefore, if precision is an issue, the number of proposed hypernyms should be limited.

Figure 6.10 has the term "airplane" and its hyponyms. Here, the graph is only nearly complete. As a result, the terms "airplane" and "biplane" are tied for the top betweenness score, shown in parenthesis, with betweenness 1.0. Figure 6.11 plots the graph for the hypernym "chicken". The hyponym "broiler" is tied with "chicken" with a betweenness score of 1.0. An abstract example is "disadvantage", shown in fig. 6.12. In this plot, both "disadvantage" and "loss" have centrality 7.0.

Naturally, when we use term frequency as a tiebreak, ties become much less of an issue. Table 6.3 lists the accuracy scores when we use term frequency to break ties. Here, the scores for graphs that have ties, in the last column, are higher than the overall scores. This shows that frequency assisted centrality is more useful than centrality alone.

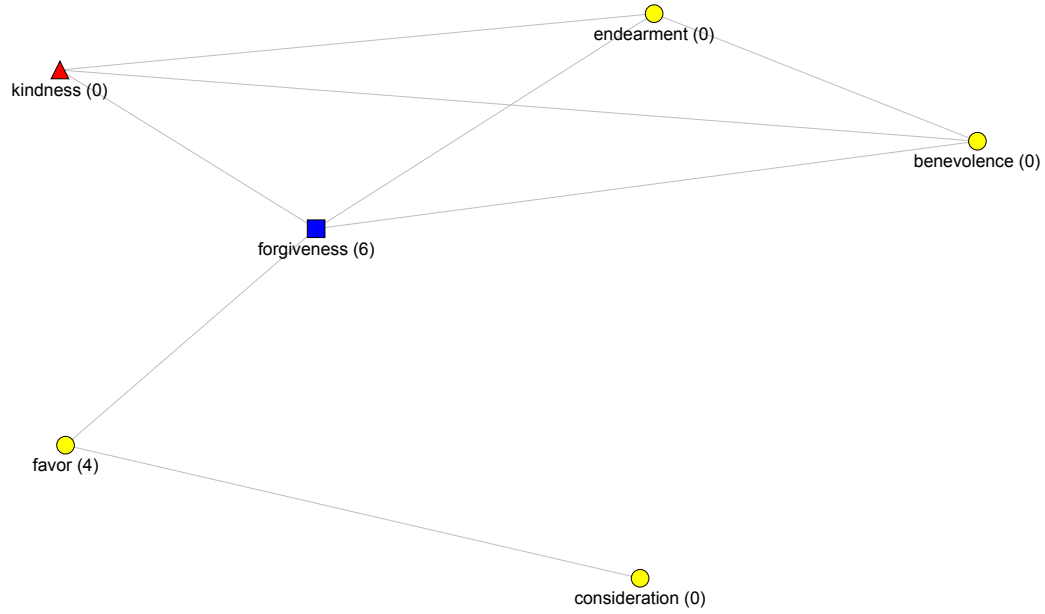
6.4.3 Different center

In many cases, the graph simply has a center different from the hypernym. This happens with both concrete and abstract terms. Figure 6.13 shows an example with abstract terms. Here, "forgiveness" is more central than the hypernym "kindness". Again, the centrality scores are shown in parenthesis in the plot. Similarly, fig. 6.14 shows the plot for the abstract term "rejection". Figure 6.15 displays a graph where the concrete hypernym "larva" has centrality 0.0, while "caterpillar" has centrality 5.0. Figure 6.16 plots the graph for the term "cloud".

6.5 Summary

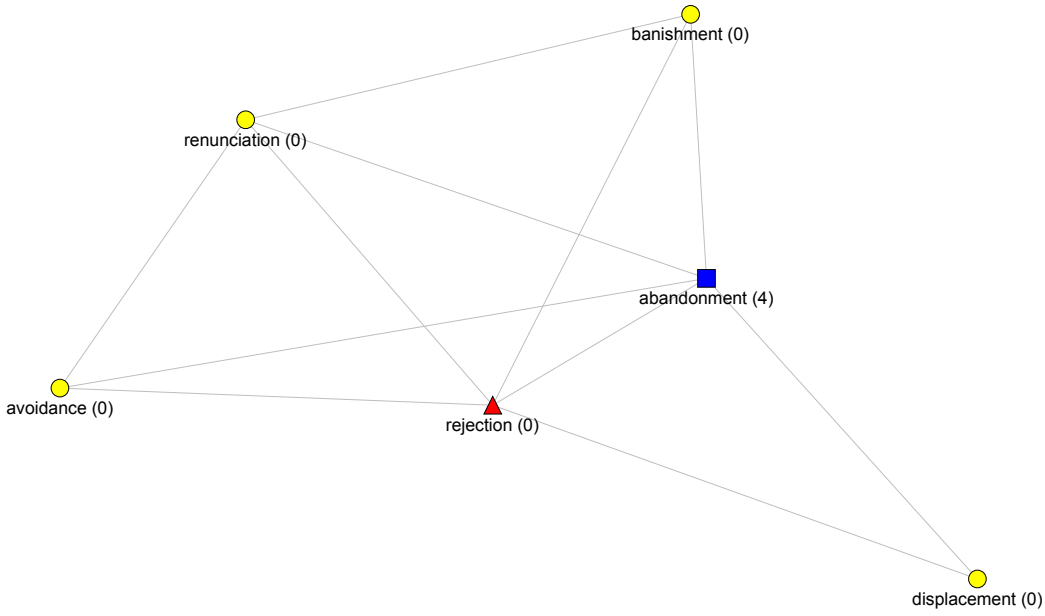
The results show that there is some correspondence between centrality and hypernymy in our hypernym/hyponym graphs. The hypernym is the most central term in at best around 32% of the cases. This suggests that centrality alone is insufficient to determine hypernymy. However, when we additionally use term frequency to break ties in centrality, the results are better with accuracy around 46%.

All our centrality measures consistently beat the random baseline. With ties broken arbitrarily, only betweenness beats the stronger frequency baseline. When term frequency is utilized, both betweenness and degree centrality beat the frequency baseline by a substantial margin. Our results are consistent across the two models and corpora used. We can therefore expect similar performance with other comparable models and corpora.



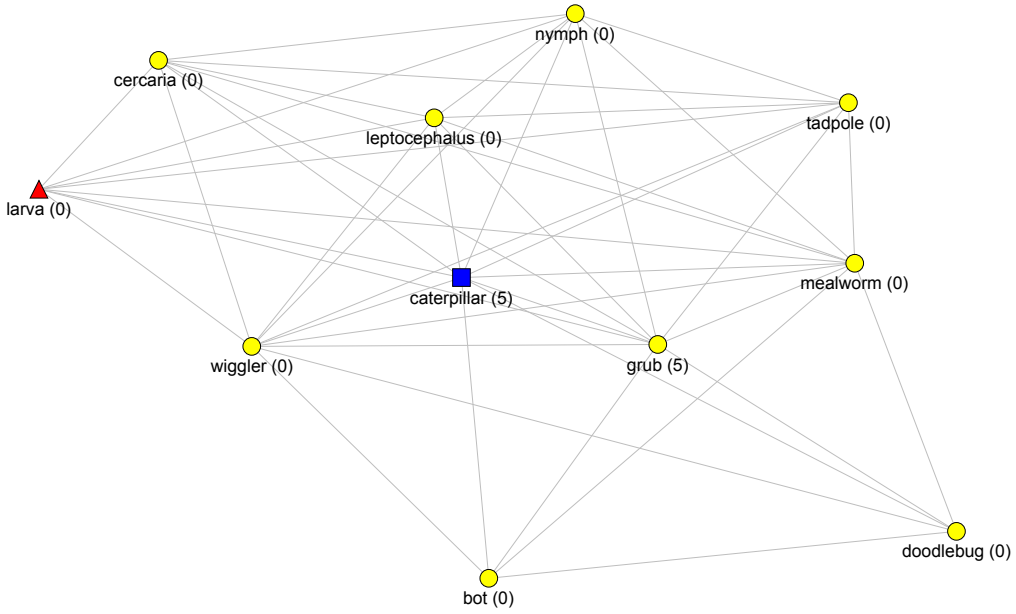
Red triangle: hypernym, blue square: highest betweenness

Figure 6.13: The hypernym “kindness” and its WordNet hyponyms



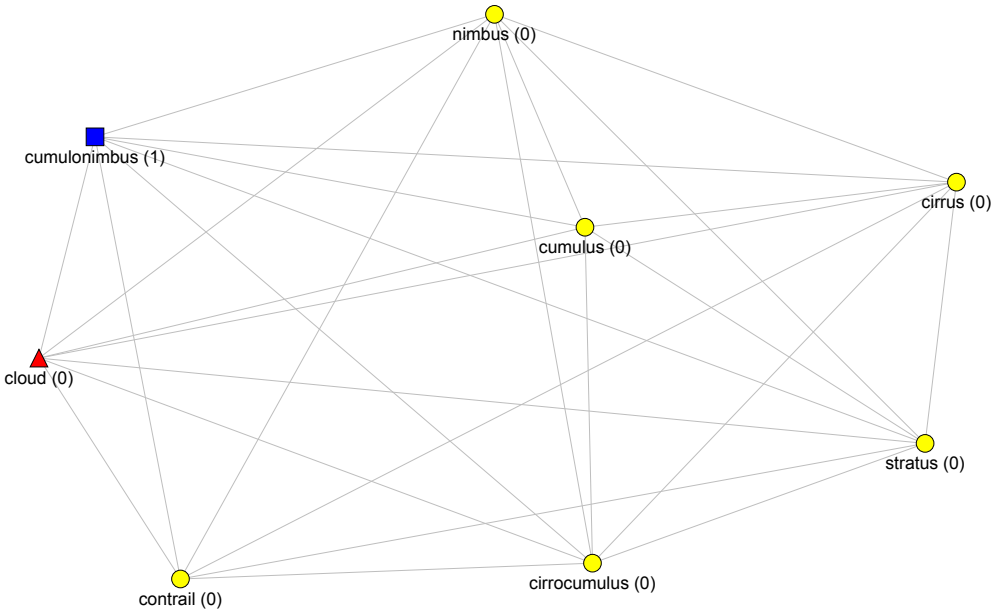
Red triangle: hypernym, blue square: highest betweenness

Figure 6.14: The hypernym “rejection” and its WordNet hyponyms



Red triangle: hypernym, blue square: highest betweenness

Figure 6.15: The hypernym “larva” and its WordNet hyponyms



Red triangle: hypernym, blue square: highest betweenness

Figure 6.16: The hypernym “cloud” and its WordNet hyponyms

Depending on the requirements of a specific application, even 46% accuracy might not be sufficient. In such cases, graph centrality might be useful as one of many features in a larger hypernym induction system. Our results suggest that such a feature extraction algorithm should employ the degree centrality measure, using term frequency to break ties. An appropriate threshold seems to be in the range from 0.2 to 0.5.

However, to use this method, either alone or as part of a larger system, we would need a quite small neighborhood graph containing hypernym and hyponym candidates. Selecting the terms that are relevant to include in this neighborhood graph is another substantial task in itself.

As extrinsic evaluation of our converted graph models, these results seem to agree with the results in the previous chapters, both in terms of performance and optimum threshold. The intrinsic evaluation showed that our converted models could yield performance within 80% to 94% of the performance of the original model. In this chapter, we have determined that we can use our converted models for hypernym discovery. This tells us that our converted models retains information that is useful for this task.

While the optimum threshold varies between the different centrality measures, the maxima tend to lie in the range between 0.2 and 0.5. This result is consistent with the evaluation of the threshold method in the previous two chapters. In both the intrinsic evaluation and the evaluation by word sense induction, we found it beneficial to drop some edges. In both cases, the optimal threshold was in the range from 0.2 to 0.5. In summary, we have demonstrated that we can use our converted graph models for hypernym discovery. However, there still remains work to turn this into a complete hypernym discovery system.

The next chapter is the conclusion of this thesis. We summarize our results, and draw a conclusion. We also discuss possible future work.

Chapter 7

Conclusion

In this thesis, we have made and evaluated procedures for converting between different lexical semantic representations. To the best of our knowledge, this is the first work that sets out to do a comprehensive evaluation of conversion between lexical semantic vector space models and graph models. We have implemented three different methods for converting from vector space models to graph models: the threshold method, the kNN method and the variable-k method. We have also implemented a single method for converting from graph models to word embedding models. Further, we have done comprehensive evaluation of our conversion procedures and their results. We have performed extensive intrinsic evaluation using SimLex-999 (Hill, Reichart and Korhonen 2015) and other gold standard data sets from the wordvectors.org (Faruqui and Dyer 2014) evaluation suite. We have also done extrinsic evaluation of our converted graph models. We have evaluated our graph models in two real-world tasks: word sense induction/disambiguation and hypernym discovery.

The use of word embeddings has become widespread, in large part because of the increasing prevalence of deep learning methods. However, training high quality word embeddings is time-consuming and requires large corpora. Therefore, an increasing number of pre-trained word embedding models have been made available, for instance by the Nordic language processing laboratory.[1] These pre-trained word embeddings are useful in many natural language processing tasks. However, there are also many graph-based algorithms, which cannot use word embeddings directly. In this thesis, we have converted high-quality word embeddings to graph models. Thus, we have been able to use these models in graph-based algorithms. We have also evaluated our conversion results thoroughly.

7.1 Results and contributions

We have hypothesized that we can convert lexical semantic representations without significant loss. Our results support this hypothesis. Our

[1] http://vectors.nlpl.eu/repository/

evaluation shows that all three of our methods are suitable for converting vector space models to graph models. All our conversion methods leave out some potential edges, so they are certainly lossy. Nevertheless, our evaluation shows that we can achieve good results despite leaving out a vast majority of the edges. This shows that the left out, low similarity edges do not contribute information that is significant for our tasks.

We have also hypothesized that filtering out or dropping low similarity edges is advantageous. Our results in both the intrinsic and extrinsic evaluation support this. Across all tasks, graph models with a large amount of edges filtered out outperform complete graphs. This applies to both time/space use and the various evaluation metrics. We can therefore conclude that producing complete graphs is rarely desirable.

The methods have different benefits and drawbacks that make them suitable for different problems. Thus, the best conversion method and hyperparameters is task specific. The threshold method is most likely to produce graphs with a small-world structure. Some algorithms, such as HyperLex, work only with small-world graphs. Therefore, they likely require the threshold method. For the threshold method, we achieve the best evaluation results with thresholds in the range between 0.2 and 0.5. This applies to both the intrinsic and the extrinsic evaluation. However, the actual optimal value differs between different applications, and must be determined by evaluation in the actual application.

Both the kNN method and the variable-k method outperform the threshold method on the SimLex-999 intrinsic evaluation. Therefore, one of these methods is probably preferable for applications that do not require small-world graphs. The variable-k method stores less information about infrequent terms, and is therefore unlikely to be optimal in applications where infrequent terms are important. For both these methods, the optimum value of k varies with the model size. Therefore, an appropriate value should be chosen by doing an evaluation in the intended application.

When developing an application based on converting vector space models to graphs, we should evaluate all three conversion methods with a wide range of parameters. This way, we ensure a thorough exploration of the parameter space. However, this requires a performance metric for the application.

We have also evaluated our procedure for converting graph models to word embeddings. Our conversion method consists of simply using the adjacency matrix as a vector model. We perform dimensionality reduction on the adjacency matrix to obtain our converted word embeddings. While there are many other methods based on the adjacency matrix, as discussed in section 3.1.3, this simple approach does not seem common. However, our method only works with weighted graphs. This conversion has only one parameter, which is the dimensionality of the word embeddings. This conversion is therefore simple to evaluate. We have kept the dimensionality fixed at 300, which is the size of the pre-trained word embeddings we have used for evaluation. While it would have been interesting to evaluate the effect of the vector dimensionality as well, we deemed that to be outside the scope of this thesis. We have only evaluated this procedure as part of the

intrinsic evaluation, with SimLex-999 and the wordvectors.org evaluation suite. Our intrinsic evaluation has shown that this conversion method works well. We have obtained scores within 80% to 94% of the original models with both SimLex-999 and most of the other gold data sets. These scores are for the full evaluation, converting from word embeddings to a graph model, and back to word embeddings. Therefore, some of the loss can be attributed to the first conversion from word embeddings to graph model. These results suggest that the conversion loss is small in both conversion steps.

We have performed extrinsic evaluation of our converted graph models in two real-world tasks. The results of the extrinsic evaluation agree with the intrinsic evaluation. We have demonstrated that our graph models are suitable for use in both word sense induction and hypernym discovery. We have successfully used HyperLex for word sense induction based on our graph models. We have also demonstrated that spinglass clustering can be used for word sense induction, outperforming our implementation of HyperLex.

We have made the following three contributions to the field of natural language processing in this thesis:

Conversion methods and evaluation. Our first contribution is the software procedures for converting between different lexical semantic representations. Our procedures implement three methods for converting from vector space models to graph models, and one method for converting the other way. All the project software is freely available on GitHub.[2] Additionally, we have contributed a comprehensive evaluation of these conversion procedures. We have done a thorough intrinsic evaluation of our conversion in both directions. For the extrinsic evaluation, we have focused on the conversion from word embeddings to graph models. These conversion procedures and the evaluation should be useful in projects involving a combination of vector and graph lexical semantic models.

Word sense induction method. Our method for word sense induction is another contribution. Traditionally, HyperLex has been used with co-occurrence graphs based on small corpora made from contexts of the target word (Véronis 2004). Therefore, the co-occurrence graphs only contain semantic information from these limited contexts. We have used HyperLex and spinglass clustering with graphs converted from high quality word embeddings trained on large, diverse corpora. We have shown that when these word embeddings are converted to a graph model, the result can be a good representation of the target term and its context. Our system outperformed seven of the nine participants in the WSI part of SemEval-2013 task 11 (Navigli and Vannella 2013).

Hypernym discovery method. Our last contribution is our experimental method of hypernym discovery. We have shown that graph cent-

[2] https://github.com/ewinge/converter

rality can indicate hypernymy. This can be useful as a feature in a larger hypernym discovery or taxonomy extraction system. Our method requires a small graph of candidate terms, but it should be possible to develop a method for constructing such graphs from a text.

7.2 Future work

We have made three different procedures for converting vector space models. However, it might be possible to better utilize local structure in the vector space. It would be beneficial to vary the number of edges depending on the density of the vector space. With the threshold method, this happens automatically, to some extent. Dense regions of the vector space will have nodes that are closer, and thus are more likely to have similarity above the threshold. However, with the kNN method all nodes get the same number of neighbors or edges. If we could use higher values of k for dense regions of the vector space, and lower for sparse regions, the resulting graph would better reflect the VSM. This should also yield graphs with a more small-world like structure.

Further, our word sense induction method can probably be improved. It would be useful to support multi-word entities in the algorithm. Our current approach only supports multi-word entities that exist as embeddings in the word embedding model. We can add support for combining terms into multi-word entities either by vector arithmetic before the conversion, or by merging nodes in the graph model. Different implementations of these two approaches should be evaluated to determine the best method.

Our hypernym discovery method should also be explored further. Hypernym discovery is a challenging but important task. It is useful in both information retrieval and artificial intelligence. While we show that it is possible to use centrality to predict hypernyms, we have not developed a way to apply this to text. To use our method on a text, we must extract relevant candidate nouns from the text. Different methods of selecting candidate nouns and the size of the context to look in should be evaluated.

It would also be interesting to combine our hypernym discovery method with other graph-based methods for taxonomy extraction. For example, Jana and Goyal (2018) employ five other graph measures for co-hyponymy detection. They use graph measures such as shortest path between nodes and edge density in neighborhoods in a supervised classifier. While their method is based on co-occurrence graphs, not word embeddings, it would be interesting to adopt this method to our converted graph models. In conclusion, there are many opportunities for future research following up on the results in this thesis.

Bibliography

Agirre, Eneko, Enrique Alfonseca et al. (2009). 'A Study on Similarity and Relatedness Using Distributional and WordNet-Based Approaches'. In: *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics*. NAACL '09. Boulder, Colorado, pp. 19–27.

Agirre, Eneko, David Martínez et al. (2006). 'Two Graph-Based Algorithms for State-of-the-Art WSD'. In: *Proceedings of the 2006 Conference on Empirical Methods in Natural Language Processing*. EMNLP '06. Sydney, Australia, pp. 585–593.

Baker, Simon, Roi Reichart and Anna Korhonen (2014). 'An Unsupervised Model for Instance Level Subcategorization Acquisition'. In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*. Doha, Qatar, pp. 278–289.

Barabási, Albert-László and Réka Albert (1999). 'Emergence of Scaling in Random Networks'. In: *Science* 286.5439, pp. 509–512.

Baroni, Marco, Georgiana Dinu and Germán 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)*. Baltimore, Maryland, pp. 238–247.

Bengio, Yoshua et al. (2003). 'A Neural Probabilistic Language Model'. In: *Journal of machine learning research* 3 (Feb), pp. 1137–1155.

Bernier-Colborne, Gabriel and Caroline Barriere (2018). 'CRIM at SemEval-2018 Task 9: A Hybrid Approach to Hypernym Discovery'. In: *Proceedings of The 12th International Workshop on Semantic Evaluation*. New Orleans, Louisiana, pp. 725–731.

Biemann, Chris (2016). 'Vectors or Graphs? On Differences of Representations for Distributional Semantic Models'. In: *Proceedings of the 5th Workshop on Cognitive Aspects of the Lexicon (CogALex-V)*. The 26th International Conference on Computational Linguistics (COLING 2016). Osaka, Japan, pp. 1–7.

Bird, Steven and Edward Loper (2004). 'NLTK: The Natural Language Toolkit'. In: *Proceedings of the ACL 2004 on Interactive Poster and Demonstration Sessions*. ACLdemo '04. Barcelona, Spain.

Bojanowski, Piotr et al. (2017). 'Enriching Word Vectors with Subword Information'. In: *Transactions of the Association for Computational Linguistics* 5, pp. 135–146.

Bordea, Georgeta, Paul Buitelaar et al. (2015). 'SemEval-2015 Task 17: Taxonomy Extraction Evaluation (TExEval)'. In: *Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval 2015)*. Denver, Colorado, pp. 902–910.

Bordea, Georgeta, Els Lefever and Paul Buitelaar (2016). 'Semeval-2016 Task 13: Taxonomy Extraction Evaluation (Texeval-2)'. In: *Proceedings of the 10th International Workshop on Semantic Evaluation (SemEval-2016)*. San Diego, California, USA, pp. 1081–1091.

Brown, Peter F. et al. (1992). 'Class-Based N-Gram Models of Natural Language'. In: *Computational Linguistics* 18.4, pp. 467–479.

Bruni, Elia et al. (2012). 'Distributional Semantics in Technicolor'. In: *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics*. ACL '12. Jeju, Republic of Korea, pp. 136–145.

Camacho-Collados, Jose et al. (2018). 'SemEval-2018 Task 9: Hypernym Discovery'. In: *Proceedings of The 12th International Workshop on Semantic Evaluation*. New Orleans, Louisiana, pp. 712–724.

Cao, Shaosheng, Wei Lu and Qiongkai Xu (2015). 'GraRep: Learning Graph Representations with Global Structural Information'. In: *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*. CIKM '15. New York, NY, USA, pp. 891–900.

Cao, Shaosheng, Wei Lu and Qiongkai Xu (2016). 'Deep Neural Networks for Learning Graph Representations'. In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. AAAI'16. Phoenix, Arizona, pp. 1145–1152.

Church, Kenneth Ward and Patrick Hanks (1989). 'Word Association Norms, Mutual Information, and Lexicography'. In: *Proceedings of the 27th Annual Meeting on Association for Computational Linguistics*. ACL '89. Vancouver, British Columbia, Canada, pp. 76–83.

Csardi, Gabor and Tamas Nepusz (2006). 'The Igraph Software Package for Complex Network Research'. In: *InterJournal, Complex Systems* 1695.5, pp. 1–9.

Deerwester, Scott et al. (1990). 'Indexing by Latent Semantic Analysis'. In: *Journal of the American Society for Information Science* 41.6, pp. 391–407.

Fares, Murhaf et al. (2017). 'Word Vectors, Reuse, and Replicability: Towards a Community Repository of Large-Text Resources'. In: *Proceedings of the 21st Nordic Conference on Computational Linguistics, NoDaLiDa*. Gothenburg, Sweden, pp. 271–276.

Faruqui, Manaal and Chris Dyer (2014). 'Community Evaluation and Exchange of Word Vectors at Wordvectors.Org'. In: *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*. Baltimore, USA.

Finkelstein, Lev et al. (2002). 'Placing Search in Context: The Concept Revisited'. In: *ACM Transactions on Information Systems* 20.1, pp. 116–131.

Firth, J. R. (1935). 'The Technique of Semantics.' In: *Transactions of the Philological Society* 34.1, pp. 36–73.

Firth, J. R. (1957). 'A Synopsis of Linguistic Theory 1930-55.' In: *Studies in Linguistic Analysis*, pp. 1–32.

Gliozzo, Alfio et al. (2013). 'JoBimText Visualizer: A Graph-Based Approach to Contextualizing Distributional Similarity'. In: *Proceedings of TextGraphs@EMNLP 2013: The 8th Workshop on Graph-Based Methods for Natural Language Processing*. Seattle, Washington, USA, pp. 6–10.

Goldberg, Yoav (2016). 'A Primer on Neural Network Models for Natural Language Processing'. In: *Journal of Artificial Intelligence Research* 57.1, pp. 345–420.

Grover, Aditya and Jure Leskovec (2016). 'Node2Vec: Scalable Feature Learning for Networks'. In: *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. New York, NY, USA, pp. 855–864.

Gutmann, Michael and Aapo Hyvärinen (–0013–2010). 'Noise-Contrastive Estimation: A New Estimation Principle for Unnormalized Statistical Models'. In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Vol. 9. Proceedings of Machine Learning Research. Sardinia, Italy, pp. 297–304.

Gyllensten, Amaru Cuba and Magnus Sahlgren (2015). 'Navigating the Semantic Horizon Using Relative Neighborhood Graphs'. In: *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. Lisbon, Portugal, pp. 2451–2460.

Halawi, Guy et al. (2012). 'Large-Scale Learning of Word Relatedness with Constraints'. In: *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '12. Beijing, China, pp. 1406–1414.

Halko, Nathan, Per-Gunnar Martinsson and Joel A. Tropp (2011). 'Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions'. In: *SIAM Review* 53.2, pp. 217–288.

Hamilton, Will, Zhitao Ying and Jure Leskovec (2017). 'Inductive Representation Learning on Large Graphs'. In: *Advances in Neural Information Processing Systems 30*. Long Beach, California, USA, pp. 1024–1034.

Hamilton, William L., Rex Ying and Jure Leskovec (2017). 'Representation Learning on Graphs: Methods and Applications'. In: arXiv: 1709.05584 [cs].

Harris, Zellig (1954). 'Distributional Structure'. In: *Word* 10.23, pp. 146–162.

Hearst, Marti A. (1992). 'Automatic Acquisition of Hyponyms from Large Text Corpora'. In: *COLING 1992: Proceedings of the 15th International Conference on Computational Linguistics, Volume 2*. Nantes, France.

Hill, Felix, Roi Reichart and Anna Korhonen (2015). 'SimLex-999: Evaluating Semantic Models With (Genuine) Similarity Estimation'. In: *Computational Linguistics* 41.4, pp. 665–695.

Hubert, Lawrence and Phipps Arabie (1985). 'Comparing Partitions'. In: *Journal of Classification* 2.1, pp. 193–218.

Jaccard, Paul (1901). 'Étude Comparative de La Distribution Florale Dans Une Portion Des Alpes et Des Jura'. In: *Bulletin del la Société Vaudoise des Sciences Naturelles* 37, pp. 547–579.

Jana, Abhik and Pawan Goyal (2018). 'Network Features Based Co-Hyponymy Detection'. In: *Proceedings of the Eleventh International Conference on Language Resources and Evaluation*. Miyazaki, Japan.

Kanerva, P., J. Kristofersson and A. Holst (2000). 'Random Indexing of Text Samples for Latent Semantic Analysis'. In: *Proceedings of the 22nd Annual Conference of the Cognitive Science Society*. Vol. 22. Erlbaum, New Jersey, p. 1036.

Kilgarriff, A. and M. Palmer (2000). 'Introduction to the Special Issue on SENSEVAL'. In: *Computers and the Humanities* 34.1, pp. 1–13.

Lee, Daniel D. and H. Sebastian Seung (1999). 'Learning the Parts of Objects by Non-Negative Matrix Factorization'. In: *Nature* 401.6755, p. 788.

Lesk, Michael (1986). 'Automatic Sense Disambiguation Using Machine Readable Dictionaries: How to Tell a Pine Cone from an Ice Cream Cone'. In: *Proceedings of the 5th Annual International Conference on Systems Documentation*. SIGDOC '86. Toronto, Ontario, Canada, pp. 24–26.

Luong, Thang, Richard Socher and Christopher Manning (2013). 'Better Word Representations with Recursive Neural Networks for Morphology'. In: *Proceedings of the Seventeenth Conference on Computational Natural Language Learning*. Sofia, Bulgaria, pp. 104–113.

Manandhar, Suresh et al. (2010). 'SemEval-2010 Task 14: Word Sense Induction & Disambiguation'. In: *Proceedings of the 5th International Workshop on Semantic Evaluation*. Uppsala, Sweden, pp. 63–68.

Manning, Christopher D. (1999). *Foundations of Statistical Natural Language Processing*. In collab. with Hinrich Schütze. Cambridge, Mass: MIT Press.

Manning, Christopher et al. (2014). 'The Stanford CoreNLP Natural Language Processing Toolkit'. In: *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*. Baltimore, Maryland, pp. 55–60.

Mikolov, Tomas, Kai Chen et al. (2013). 'Efficient Estimation of Word Representations in Vector Space'. In: arXiv: 1301.3781 [cs].

Mikolov, Tomas, Ilya Sutskever et al. (2013). 'Distributed Representations of Words and Phrases and Their Compositionality'. In: *Advances in Neural Information Processing Systems 26*. Lake Tahoe, Nevada, USA, pp. 3111–3119.

Milgram, Stanley (1967). 'The Small World Problem'. In: *Psychology Today* 1.1, pp. 60–67.

Miller, George A. (1995). 'WordNet: A Lexical Database for English'. In: *Communications of the ACM* 38.11, pp. 39–41.

Miller, George A. and Walter G. Charles (1991). 'Contextual Correlates of Semantic Similarity'. In: *Language and Cognitive Processes* 6.1, pp. 1–28.

Navigli, Roberto (2012). 'A Quick Tour of Word Sense Disambiguation, Induction and Related Approaches'. In: *Proceedings of the 38th International Conference on Current Trends in Theory and Practice of Computer Science*. SOFSEM'12. Špindlerův Mlýn, Czech Republic, pp. 115–129.

Navigli, Roberto and Giuseppe Crisafulli (2010). 'Inducing Word Senses to Improve Web Search Result Clustering'. In: *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*. Cambridge, MA, pp. 116–126.

Navigli, Roberto and Daniele Vannella (2013). 'SemEval-2013 Task 11: Word Sense Induction and Disambiguation within an End-User Application'. In: *Second Joint Conference on Lexical and Computational Semantics (*SEM), Volume 2: Proceedings of the Seventh International Workshop on Semantic Evaluation (SemEval 2013)*. Atlanta, Georgia, USA, pp. 193–201.

Osgood, Charles E., George J. Suci and Percy H. Tannenbaum (1957). *The Measurement of Meaning*. Urbana, Ill: University of Illinois Press.

Ou, Mingdong et al. (2016). 'Asymmetric Transitivity Preserving Graph Embedding'. In: *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. New York, NY, USA, pp. 1105–1114.

Parker, Robert et al. (2011). *English Gigaword Fifth Edition LDC2011T07*. Philadelphia: Linguistic Data Consortium.

Pedregosa, F. et al. (2011). 'Scikit-Learn: Machine Learning in Python'. In: *Journal of Machine Learning Research* 12, pp. 2825–2830.

Pennington, Jeffrey, Richard Socher and Christopher Manning (2014). 'Glove: Global Vectors for Word Representation'. In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*. Doha, Qatar, pp. 1532–1543.

Perozzi, Bryan, Rami Al-Rfou and Steven Skiena (2014). 'DeepWalk: Online Learning of Social Representations'. In: *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '14. New York, NY, USA, pp. 701–710.

Piña, Luis Nieto and Richard Johansson (2016). 'Embedding Senses for Efficient Graph-Based Word Sense Disambiguation'. In: *Proceedings of the 2016 Workshop on Graph-Based Methods for Natural Language Processing, NAACL-HLT 2016*. San Diego, California, USA, pp. 1–5.

Radinsky, Kira et al. (2011). 'A Word at a Time: Computing Word Relatedness Using Temporal Semantic Analysis'. In: *Proceedings of the 20th International Conference on World Wide Web*. WWW '11. Hyderabad, India, pp. 337–346.

Rand, William M. (1971). 'Objective Criteria for the Evaluation of Clustering Methods'. In: *Journal of the American Statistical Association* 66.336, pp. 846–850. JSTOR: 2284239.

Řehůřek, Radim and Petr Sojka (2010). 'Software Framework for Topic Modelling with Large Corpora'. English. In: *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. Valletta, Malta, pp. 45–50.

Reichardt, Joerg and Stefan Bornholdt (2006). 'Statistical Mechanics of Community Detection'. In: *Physical Review E* 74.1. arXiv: cond-mat/0603718.

Roller, Stephen, Douwe Kiela and Maximilian Nickel (2018). 'Hearst Patterns Revisited: Automatic Hypernym Detection from Large Text Corpora'. In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Melbourne, Australia, pp. 358–363.

Rong, Xin (2014). 'Word2vec Parameter Learning Explained'. In: arXiv: 1411.2738 [cs].

Rubenstein, Herbert and John B. Goodenough (1965). 'Contextual Correlates of Synonymy'. In: *Communications of the ACM* 8.10, pp. 627–633.

Steyvers, Mark and Joshua B. Tenenbaum (2005). 'The Large-Scale Structure of Semantic Networks: Statistical Analyses and a Model of Semantic Growth'. In: *Cognitive science* 29.1, pp. 41–78.

Turney, P. D. and P. Pantel (2010). 'From Frequency to Meaning: Vector Space Models of Semantics'. In: *Journal of Artificial Intelligence Research* 37, pp. 141–188.

Ustalov, Dmitry et al. (2018). 'Unsupervised Semantic Frame Induction Using Triclustering'. In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Melbourne, Australia, pp. 55–62.

Van der Maaten, L. J. P. and G. E. Hinton (2008). 'Visualizing High-Dimensional Data Using t-SNE'. In: *Journal of Machine Learning Research* 9 (nov), pp. 2579–2605.

Van Rijsbergen, C.J. (1979). *Information Retrieval*. 2nd ed. London: Butterworths.

Véronis, Jean (2004). 'HyperLex: Lexical Cartography for Information Retrieval'. In: *Computer Speech and Language* 18, pp. 223–252.

Walt, Stéfan van der, S. Chris Colbert and Gaël Varoquaux (2011). 'The NumPy Array: A Structure for Efficient Numerical Computation'. In: *Computing in Science & Engineering* 13.2, pp. 22–30.

Wang, Chengyu, Xiaofeng He and Aoying Zhou (2017). 'A Short Survey on Taxonomy Learning from Text Corpora: Issues, Resources and Recent Advances'. In: *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. Copenhagen, Denmark, pp. 1190–1203.

Wang, Daixin, Peng Cui and Wenwu Zhu (2016). 'Structural Deep Network Embedding'. In: *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. New York, NY, USA, pp. 1225–1234.

Watts, Duncan J. and Steven H. Strogatz (1998). 'Collective Dynamics of 'Small-World' Networks'. In: *Nature* 393.6684, pp. 440–442.

Wittgenstein, Ludwig (1953). *Philosophical Investigations*. Translated by G.E.M. Anscombe. Oxford: Blackwell.

Yang, Dongqiang and David M. W. Powers (2006). 'Verb Similarity on the Taxonomy of Wordnet'. In: *Proceedings of the Third International WordNet Conference*. South Jeju Island, Korea.