Computing Distributed Representations for Polysemous Words

Master Thesis
Software Systems Engineering

Haiqing Wang
Matriculation number 340863

July 4, 2016

Supervisors:
Prof. Dr. Gerhard Lakemeyer
Prof. Dr. Christian Bauckhage

Advisors:
Dr. Gerhard Paaß
Dr. Jörg Kindermann

Acknowledgements

You should add some acknowledgements: thank your advisors, supervisors, parents, family, colleagues, students, etc.

Abstract

You can add an abstract if you like.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The following points should appear in the abstract and in more elaborate form in the introduction:

1. Machine Learning
2. Text Analytics: detect word sense
3. Sense embeddings to represent word senses
4. Polysemy: Multiple senses of a word.
5. What is the best way to do this? → multiple senses per word → investigate and improve current methods with multiple senses per word
6. Main Task of the thesis: Implementation of methods with multiple senses per word in Spark to be able to execute in parallel
7. Train with Wikipedia corpus. Evaluate by inspecting similar word senses. and evaluate similarity tasks.
8. Improvement: better speed and better similarity
9. organization of the thesis

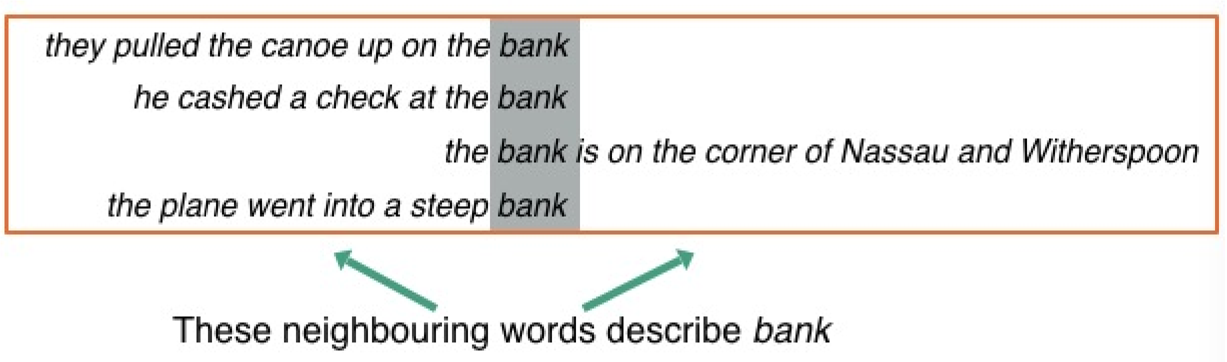
1.1 Distributed word representations

Machine learning approaches for natural language processing have to represent the words of a language in a way such that Machine Learning modules may process them. This is especially important for text mining, where data mining modules analyze text corpora.
Traditional text mining analyses use the vector space representation [Salton et al., 1975], where a word is represented by a sparse vector of the size of the vocabulary (usually more than 100.000), where all values are 0 except the entry for the actual word. This

representation is also called *One-hot representation*. This sparse representation, however, has no information on the semantic similarity of words.

Recently word representations have been developed which represent each word as a vector of k (e.g. $k = 100$) real numbers as proposed by [Collobert and Weston, 2008] and [Mikolov et al., 2013]. Generally, we call such a vector a *word embedding*. By using a large corpus in an unsupervised algorithm word representations may be derived such that words with similar syntax and semantics have representations with a small Euclidean distance. Hence the distances between word embeddings corresponds to the semantic similarity of underlying words. These embeddings may be visualized to show comunalities and differences between words, sentences and documents. Subsequently these word representations may be employed for further text mining analyses like *opinion mining* [Socher et al., 2013], Kim 2014, Tang et al. 2014) or *semantic role labeling* [Zhou and Xu, 2015] which benefit from this type of representation [Collobert et al., 2011].

These algorithms are based on the very important assumption that if the contexts of two words are similar, their representations should be similar as well [Harris, 1954].[1] Figure 1.1 shows how neighboring words determine the sense of the word "bank" in a number of example sentences. So many actual text mining methods make use of the context of words to generate embeddings.



they pulled the canoe up on the bank

he cashed a check at the bank

the bank is on the corner of Nassau and Witherspoon

the plane went into a steep bank

These neighbouring words describe *bank*

Figure 1.1: Neigboring words defining the specific sense of "bank".

A traditional approach to derive word embeddings is the analysis of the word co-occurrence matrix [Deerwester et al., 1990]. It is based on the one-hot representation. Each row of the matrix represents the information of one word's context, which is sparse and huge. This matrix is decomposed using singular value decomposition (SVD) to generate low-dimension embedding vectors. The context can be the occurrences of words in the corresponding document or be the average occurrence's of surrounding words from all documents[2].

Recently, artificial neural networks became very popular to generate lower-dimensional word embeddings. Prominent algorithms are *Senna* [Collobert and Weston, 2008], *Word2vec* [Mikolov et al., 2013] and Glove [Pennington et al., 2014]. They all use randomly initialized

[1] COMMENT: please more details on this

[2] COMMENT: I do not understand that

vectors to represent words . Subsequently these embeddings are modified in such a way that the word embeddings of the neigboring words may be predicted with minimal error by a simple neural network function.

1.2 Polysemy

Note that in the approaches described above each word is mapped to a single embedding vector. It is well known, however, that a word may have several different meanings, i.e. is *polysemous*. For example the word "bank" among others may designate:

- the slope beside a body of water,
- a financial institution,
- a flight maneuver of an airplane.

Further examples of polysemy are the words "book", "milk" or "crane". WordNet [Fellbaum, 1998] and other lexical resources show, that most common words have 3 to 10 different meanings. Obviously each of these meanings should be represented by a separate embedding vector, otherwise the embedding will no longer represent the underlying sense. This in addition will harm the performance of subsequent text mining analyses. Therefore we need methods to learn embeddings for senses rather than words.
Sense embeddings are a refinement of word embeddings. For example, "bank" can appear either together with "money", "account", "check" or in the context of "river", "water", "canoe". And the embeddings of "money", "account", "check" will be quite different from the embeddings of "river", "water", "canoe". Consider the following two sentences

- They pulled the canoe up the bank.
- He cashed a check at the bank.

The word "bank" in the first sentence has a different sense than the word "bank" in the second sentence. Obviously, the context is different. So if we have a methods to determine the difference of the context, we can relabel the word "bank" to the word senses "bank$_1$" or "bank$_2$" denoting the slope near a river or the financial institution respectively. We call the number after the word the sense labels of the word "bank". This process can be performed iteratively for each word in the corpus by evaluating its context.
[3]

In the last years a number of approaches to derive sense embeddings have been presented. Huang et al. [2012] used the clustering of precomputed one-sense word embeddings and

[3] COMMENT: An alternative representation of words is generated by topic models [Blei et al., 2003], which represent each word of a document as a finite mixture of topic vectors. The mixture weights of a word depend on the actual document. This implies that a word gets different representations depending on the context. Please elaborate

their neighborhood embeddings to define the different word senses. The resulting word senses are fixed to the corresponding word neighborhoods and their values are trained until convergence. A similar approach is described by Chen et al. [2014]. Instead of a single embedding each word is represented by a number of different sense embeddings. During each iteration of the supervised training of Senna or Word2vec for each position of the word the best fitting embedding is selected according the fitness criterion[4]. Subsequently only this embedding is trained using back-propagation. Note that during training a word may be assigned to different senses thus reflecting the training process. A related approach was proposed by Tian et al. [2014].
It turned out that the resulting embeddings get better with the size of the training corpus and an increase of the dimension of the embedding vectors. This usually requires a parallel environment for the execution of the training of the embeddings. Recently *Apache Spark* [Zaharia et al., 2010] has been presented, an opensource cluster computing framework. Spark provides the facility to utilize entire clusters with implicit data parallelism and fault-tolerance against resource problems, e.g. memory shortage. The currently available sense embedding approaches are not ready to use compute clusters, e.g. by Apache Spark.

1.3 Goal and Organization of the Thesis

The main aim of this thesis is to derive expressive word representations for different senses in an efficient way. We will investigate sense assignment models which will extend known word embedding (one sense) approaches. Our goal is to implement such a method on a compute cluster using Apache Spark to be able to process larger training corpora and employ higher-dimensional sense embedding vectors. [5] Our main work will focus on the extension of Skip-gram model [Mikolov et al., 2013] in connection to the approach of [Neelakantan et al., 2015] because these models are easy to use, very efficient and convenient to train. [6] When using the Spark big data framework, we want to gain some experience and get feedback about the advantages and disadvantages of the new techniques.
[7] In the next chapter, we will introduce relative word embedding methods and sense embedding methods. We start with the neural language model and explain the early models of word embeddings. And then we focus on the word2vec Mikolov et al. [2013] especially about Skip-gram model. There will be many mathematical details including gradient calculation. After that, we will introduce two famous sense embedding models based the above word embedding works. The chapter 3 is our model description for sense embed-

[4] COMMENT: Please reformulate sentence

[5] Our goal is not to introduce a very excellent method which can get the best sense embedding results, but to try the new model structure like sense assignment and the new software tool like distributed framework Spark to get the results reasonable and efficient.

[6] And these days, some JVM based big data frameworks like Apache Spark are more and more popular, but the relative works on neural language processing especially word embedding and sense embedding use very few about these new techniques. That's the main reason that we try to use this new technique to implement our model.

[7] COMMENT: Rewrite this if the chapters are finished!

dings. We use the spark framework to implement our model. The chapter 4 will introduce our implementation and show the experiment we did including parameter comparison and word senses visualization. At last chapter conclusion, we will analysis the advantages and disadvantages about our methods including model and implementation and give some ideas about how we can improve it in the future and what else we can do.

Chapter 2

Background and Related Works

2.1 Neural Probabilistic Language Model

[1]

[2]

This section will introduce a *neural probabilistic language model* proposed by **?**. This is one of the first models using word embeddings. So what is a *word embedding*? Generally speaking, for any word $w \in D$ in the dictionary D, one can specify a real-valued vector $v(w) \in \mathbb{R}^m$ of fixed length m, $v(w)$ called the word embedding of w, and m is the length of the word embedding. Note that the D usually contains 10.000 to one million words, whereas the size of the embedding vector is usually in the range of 50 to 1000. A further understanding about the word embeddings will be explained in the next section.

Since it is a neural probabilistic language model, it is obvious to use an neural network. Figure 2.1 shows the structure of the neural network. It includes four layers: The *Input layer*, the *Projection layer*, the *Hidden layer* and the *Outputlayer*. W and U are respectively the weight matrix between projection layer and hidden layer and the weight matrix between hidden layer and output layer, $\mathbf{p}$ and $\mathbf{q}$ are the offset vectors of respectively the hidden layer and the output layer. Further details of word embeddings will be explained in the next section. [3]

[4] When talking about the above neural network, generally we consider it as a three-layer structure as following Figure 2.2. But this thesis still use the structure of Figure 2.1. On the one hand it is easy to describe, on the other hand it is more convenient to do comparison with the network structure in word2vec.

[1] COMMENT: all quantities have to be defined and described before they are used!!!!!!!!!!

[2] COMMENT: Please start first with a simple 3-layer neural network as in https://en.wikipedia.org/wiki/Artificial_neural_network .You could also use the graphics. Then describe the neural probabilistic language model

[3] COMMENT: A neural network defines a mapping between some input and output. Anways specify the input domain and its dimesnion and the output domain! What is the meaning of the input. What is the meaning of the input?

[4] COMMENT: Please describe the three-layer network above and then describe the 4-layer network

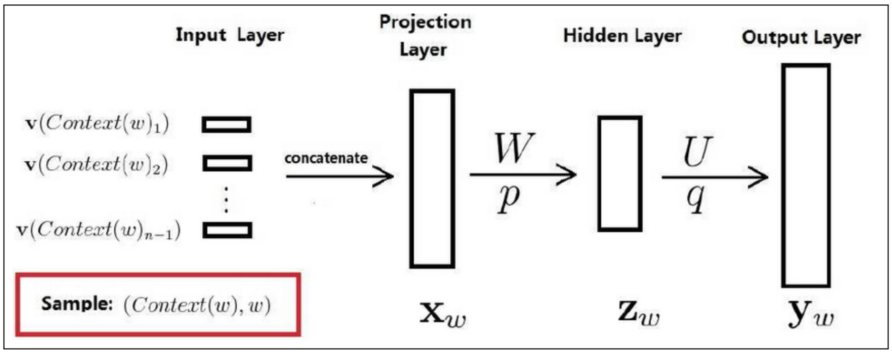


Figure 2.1: An example of 4 layers neural network.

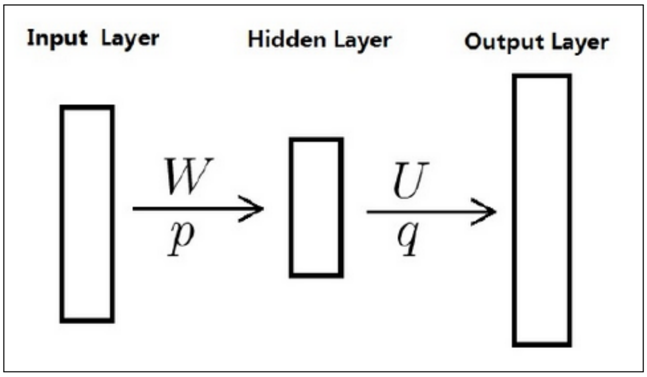


Figure 2.2: An example of 3 layers neural network

[?] also considers the connection of some neurons from projection layer and some neurons from hidden layer as Figure 2.3. Thus, there is one more weight matrix. In the numerical experiments, the author found that the introduction of the weight matrix projection layer and output layer can not improve the model effect, but it can reduce the number of training iterations.

We consider a corpus C, i.e. a collection of documents with a vocabulary D of words. [5] A document $d \in C$ consist of the words $w_1, \ldots, w_t, \ldots, w_T$ with $w_i \in D$. ... For any word w in corpus C , assuming $Context(w)$ [6] takes its front $n-1$ words (similar to the n-gram), this binary pair $(Context(w), w)$ is a a training sample.[7] So how is the sample $(Context(w), w)$ involved in computing through the neural network? Note that once word corpus C and vector length m is given, the scale of projection layer and the scale of output layer are determined. The former is $(n-1)m$, the latter is $N = |D|$, that is, the size of vocabulary. The size of the hidden layer n_h is the adjustable parameter which can be specified by the user.

Why is the size of the projected layer $(n-1)m$?[8] In fact, the input layer includes $n-1$

[5] COMMENT: What do you describe? what is the input, what are the dimensions?

[6] COMMENT: define!

[7] COMMENT: What is the task for which this is used?

[8] COMMENT: what is m

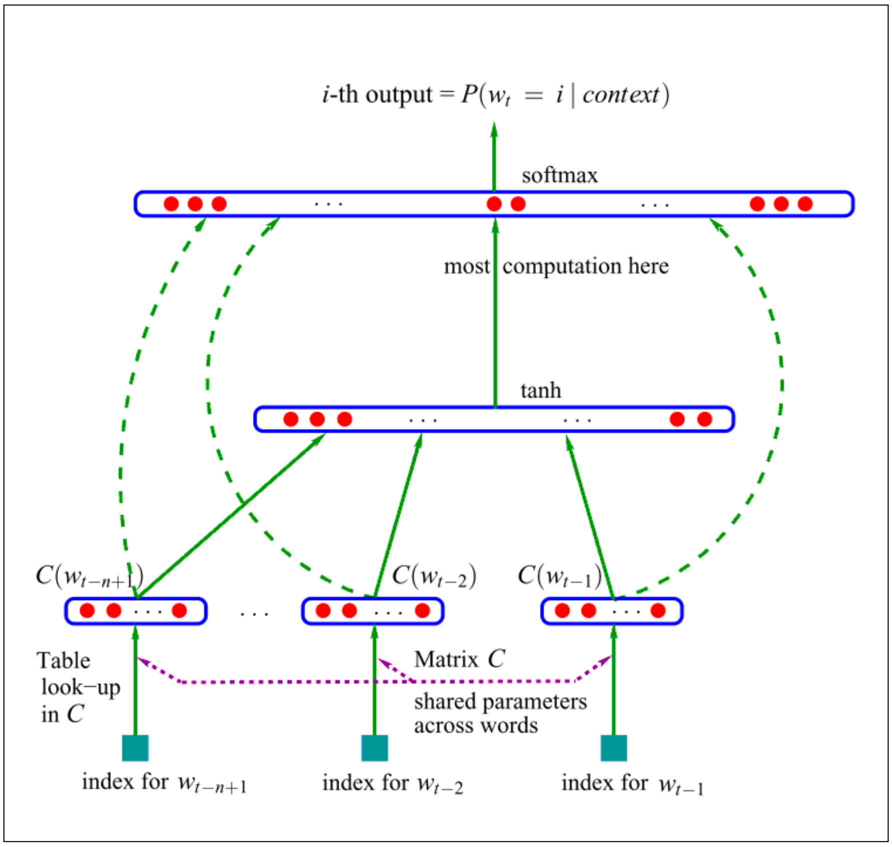


Figure 2.3: The neural network structure from [?]

words from $Context(w)$, and the vector $\mathbf{x_w}$ from projected layer is built: concatenate $n-1$ input word vectors to be a long vector whose length is $(n-1)m$. With the vector $\mathbf{x_w}$, the next calculation is clear

$$\begin{aligned} \mathbf{z_w} &= \tanh(Wx_w + \mathbf{p}), \\ \mathbf{y_w} &= Uz_w + \mathbf{q} \end{aligned} \tag{2.1}$$

where tanh is the Hyperbolic Tangent Function, used as the *Activation Function* in the hidden layer. In the above formula for a vctor $u = (u_1, \ldots, u_k)$, we definetanh$(u) = (tanh(u_1, \ldots, tanh(u_k)))$. How about the number of first few words of a given sentence is less than $n-1$? Usually, we can artificially add some filler vectors, and they will also be involved in the training process.

From the above two steps, we get [9] $\mathbf{y_w} = (y_{w,1}, y_{w,2}, \ldots, y_{w,N})^T$, which is just the vector with the length of N and its components can not represent probabilities. If you want to use $\mathbf{y_w}$'s component $y_{w,i}$ to represent that the probability of the next word is the i-th word when the context is $Context(w)$. Also you need to do a softmax normalization. After

[9] COMMENT: where comes $y_{w,i}$ form? in what set?

normalization, $p(w|Context(w))$ can be expressed as

$$p(w|Context(w)) = \frac{e^{y_{w,i_w}}}{\sum_{i=1}^{N} e^{y_{w,i}}}, \qquad (2.2)$$

where i_w represents the index of w in the dictionary D. Note that the denominator contains a term $e^{y_{w,i}}$ for every word in the vocabulary.

Formula 2.2 gives the function representation of the probability $p(w|Context(w))$, that is, it gives the function mentioned in the last section $F(w, Context(w), \theta)$. So what is the θ? In conclusion, there are two parts [10]

- Word vectors: $\mathbf{v}(w) \in \mathbb{R}^m, w \in D$ and the filter vectors
- Neural network parameters: $W \in \mathbb{R}^{n_h*(n-1)m}, \mathbf{p} \in \mathbb{R}^{n_h}; U \in \mathbb{R}^{N*n_h}, \mathbf{q} \in \mathbb{R}^N$

These parameters can be obtained by the training algorithm. On thing needs to be mentioned that, in common machine learning algorithms, the input is already known, however in the above neural probability language model shown as Figure 2.1, the input $\mathbf{v}(w)$ is not known and also needs to be training.

The next, let's look at the computation of the above model. In the above neural network, the scales of the projection layer, the hidden layer and the output layer are respectively $(n-1)m$, n_h, N, let's look at the parameters involved:[11]

1. n is the number of words contained in the context of a word, usually no more than 5
2. m is the length of word vector, usually the magnitude of $10^1 \sim 10^3$
3. n_h is specified by the user, usually not too big, like the magnitude of 10^2
4. N is the size of corpus vocabulary , related with the corpus, usually the magnitude of 10^4 to 10^6

Recombination with 2.1 and 2.2, it is not difficult to find that, the computing of entire model is mainly about the matrix-vector operations between the hidden layer and the output layer and the softmax normalization in the output layer. Therefore, many of subsequent related works are about optimization for this part, including the the work of word2vec.

Comparison with n-gram models,[12] the neural probabilistic language model mainly has the following two advantages:

[10] COMMENT: These definitions and dimensions should be given at the beginning!!! Simply state them without questions, describe their meaning and describe the model!

[11] COMMENT: These quatities have to be defined at the first usage of the associated objects

[12] COMMENT: What is an n-gram model? Has to be defined!

Similarity between words can be reflected in the word vectors. If in an (English) corpus, $S_1 =$ "A dog is running in the room" appears 10000 times, and $S_2 =$ "A cat is running in the room" only appears once. According to the n-gram model, $p(S_1)$ will certainly be much greater than $p(S_2)$.[13] Note that the only difference between S_1 and S_2 is the "dog" and "cat" , but these two words play the same role either semantically or grammatically, so $p(S_1)$ and $p(S_2)$ should be very close.
However, the probabilities $p(S_1)$ and $p(S_2)$ calculated by the neural network are approximately equal. The reason is that:

1. In the neural network probabilistic language model, there is an assumption that the "similar " words should have similar vectors [14]

2. The probability function on the word vectors is smooth, that is there is only very small influence for the probability when word vector change a little.[15]

As a result, for the following sentences

A dog is running in the room

A cat is running in the room

The cat is running in a room

A dog is walking in a bedroom

The dog was walking in the room

...

anyone appears in the corpus, the probabilities of other sentences will increase accordingly.

Models based on the word vector have smoothing already [16] (from 2.2, we can know $p(w|Context(w)) \in (0, 1)$ can not be 0), no longer need to carry the additional processing like n-gram model.
Finally, let's look back and think about what kind of role the word vector plays in the neural probability model. When training, it is just the auxiliary parameter used to construct the objective function; after the training, it seems just a by-product of the language model. However, this by-product can not be underestimated, the next section will be further elaborate its usefulness.

[13] COMMENT: Where are $p(S_i)$ defined????

[14] COMMENT: This is a consequence of its definition and not an assumption

[15] COMMENT: This is not true in general, the changes may be drastic! Elaborate the argument!

[16] COMMENT: What does that mean? What is smoothing?

2.2 Understanding of the Word Embedding

In NLP tasks, we will use machine learning algorithms to deal with natural language, but the machine can not directly understand human language, so the first thing is to transform the language to the mathematical form. How can we do such thing? Word vector provides a solution.

One of the easiest word vector is one-hot representation, which is to use a long vector to represent a word, the vector's length is N, the size of dictionary D. It only has one component which is 1, and the other components are all 0s. The position of 1 corresponds to the index of the word in dictionary. But this word vector representation has some disadvantages, such as troubled by the huge dimensionality, especially when it is applied to deep learning scenes; another thing, it can not describe the similarity between words very well. Another word vector is Distributed Representation, it was firstly proposed by **?**, which can overcome the above drawbacks from one-hot representation. The basic idea is to train the particular language to map each word into a short vector of fixed length (here "short" is respected to "long" in one-hot representation). All of these vectors constitute a vector space, and each can be regarded as a a point in the vector space. After introducing the "distance"in this space , it is possible to judge the similarity between words (morphology and syntax) according to the distance. Actually, word2vec uses this Distributed Representation for word vector.

Why is it called Distributed Representation? For one-hot representation, there is only one non-zero vector component, which is very concentrated. For Distributed Representation, vectors have a lot of non-zero components, relatively dispersed. It distributes the information of the word into many components.

Suppose that there are a different points distributed on the two-dimensional plane. For a given point, the task is to find another point closest to this point in the plane. How can we do it? Firstly, establish a Cartesian coordinate system. Based on this coordinate system, each point on which uniquely corresponds to a coordinate (x, y); and then introduce the Euclidean distance; finally calculate the distance between this point and other $a-1$ points, from which the point with the minimum distance is the one we are looking for. In the above example, the role of the coordinates (x, y) is equivalent to the word vector. It is used to mathematically quantify a point on a plane. After the coordinate system is set up, it is very easy to get the coordinate of a point. However, for NLP tasks, to get the word vector is more complex, and the word vector is not unique, which depends on the quality of the training data, training algorithm and other factors.[17]

A good word vector is valuable, for example, Ronan Collobert's team makes use of the word vector from software package SENNA ([Collobert et al., 2011]) to do part-ofspeech tagging, chunking into phrases, named entity recognition and semantic role labeling, and achieves good results. Google's Tomas Mikolov team has developed an automatic gener-

[17] COMMENT: What does this trivial stuff tell us?

ation technology for dictionary and glossary, which is able to convert one language into another language. The relation collection between words in each language, that is "language space" , can be characterized as a set of vectors in the mathematical sense. As long as the mapping and translation of a vector space to another vector space are realized, language translation can be realized. This technique has very good performance for translation between English and Spanish, with the accuracy rate up to 90%.

2.3 The model of Collobert and Weston

Collobert and Weston [2008] original main purpose is not to generate a good word vectors, or even do not want to train the language model, but to use this word vectors to complete several tasks from natural language processing, such as speech tagging, named entity recognition, phrase recognition, semantic role labeling, and so on ([Collobert and Weston, 2008] and [Collobert et al., 2011]). Due to the different purpose, their training method is also different and special. They do not use language model's idea like optimizing the probability $P(w_t|w_1, w_2, ..., w_{t-1})$, but directly use the score $f(w_{t-n+1}, ..., w_{t-1}, w_t)$ [18] to determine if the sentence is reasonable and normal; low score illustrates the sentence is not reasonable; if you put a few words randomly together, it would be certainly a negative score. The score is just about high or low, not business with probabilities.[19]
With the above assumption, Collobert and Weston used the pair-wise method to train the word vectors. Specifically, it is to minimize the following objective function.[20]

$$\sum_{x \in X} \sum_{w \in D} max\{0, 1 - f(x), f(x^{(w)})\}$$

X is the set of all consecutive n-length phrases, D is the entire dictionary. The first summation enumerates all n-length phrases from the training set, and each of them positive sample. The second summation for dictionary is to build negative samples. $x(w)$ means the phrase x replacing the middle word to the word w. In most cases, replacing the middle of the word in a normal phrase, the new phrase is certainly not the normal phrase, which is a good method to build negative sample (in most cases they are negative samples, only in rare cases the normal phrases are considered as negative samples but they would not affect the final result).

The structure of f is almost save as the network structure from [?]. The same thing is connecting n word vectors together to get a long vector and passing through one layer (a matrix multiplication) to get the hidden layer. The difference is that Collobert and Weston output layer has only one node representing the score, rather than Bengio's $|V|$ nodes.[21]

[18] COMMENT: what is f()

[19] COMMENT: the score is just a number, the larger the better. Negative / positive is not important. Please write down the equations of the model.

[20] COMMENT: you have to define f. What is input, what is output?

[21] COMMENT: What is V?

Doing so greatly reduced the computational complexity. Of course, Collobert and Weston do not want to make a real language model, but just use some idea from the language model to assist them to complete other tasks in NLP.[22]
Specifically, they give two different neural network structures window approach and sentence approach, shown as Figure 2.4 and Figure 2.5 respectively.[23] Window approach is a feedforward neural network including a linear layer, HardTanh layer. Its input is the vector concatenated by all word vectors within the current word window including itself. Window approach is able to deal with most of natural language processing tasks, but has very poor performance on SRL taks. There, they proposed sentence approach to solve such problem. It is convolutional neural network structure. Apart from the linear layer and HardTanh layer, it has another convolutional layer and Max layer.

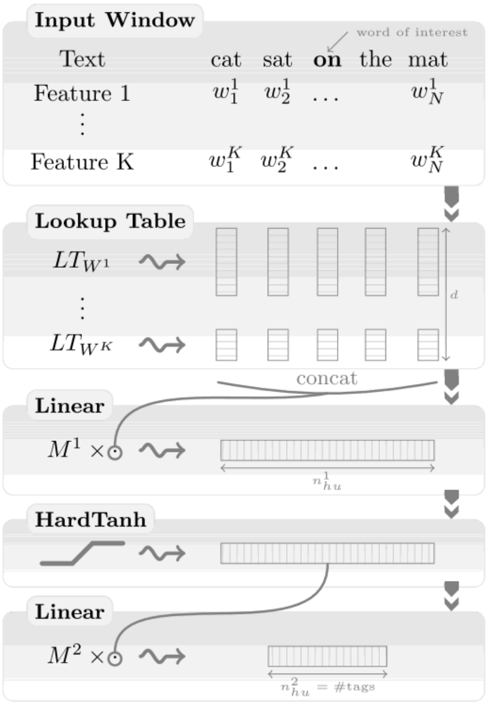


Figure 2.4: window approach(Collobert et al. [2011])

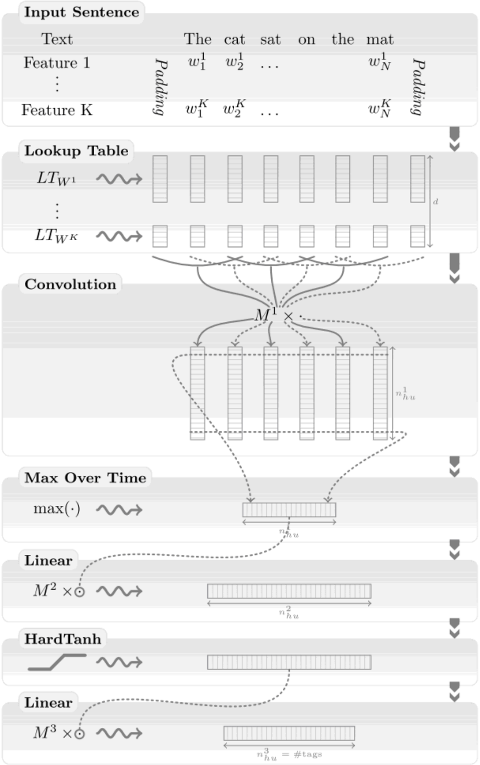


Figure 2.5: sentence approach ([Collobert et al., 2011])

In the experiment the size of window n is 11 and the size of dictionary $|V|$ is 130000.[24] They spent totally seven weeks to train word vectors from the Wikipedia English corpus

[22] COMMENT: The idea of the first paper was to get a language model!

[23] COMMENT: These models are for the subsequent NLP-tasks. Use a figure from [Collobert and Weston, 2008]

[24] COMMENT: Before the Dictionary is denoted by D. Unify this. I gave you the advice to specify a notation page, but you did not!

and Reuters corpus.

2.4 Word2Vec

This section will introduce two important model in word2vec: the *CBOW model* (Continuous Bag-of-Words Model) and the *Skip-gram model* (Continuous Skip-gram Model).[25]
The CBOW models aims to predict the current word w_t giving its context, e.g. $Context(w_t) = \{w_{t-2}, w_{t-1}, w_{t+1}, w_{t+2}\}$. Figure 2.6 of Mikolov et al. [2013] shows the schematic diagram of the CBOW model. The Skip-gram model shown in figure 2.7 has the task to predict the words of the context $Context(w_t)$ from the known current word w_t.
If we want to predict word w_t from its context the we can use the log-likelihood function of the neural network based language model (2.2) [26]

$$\mathcal{L} = \sum_{w \in \mathcal{C}} \log\ p(w|Context(w)) \tag{2.3}$$

[27] For the objective function Hierarchical Softmax CBOW word2vec model based on optimized also the form (4.1); and for the objective function based on Hierarchical Softmax of Skip-gram model, the optimization of the form

$$\mathcal{L} = \sum_{w \in \mathcal{C}} \log\ p(Context(w)|w), \tag{2.4}$$

Hence the input of the Skip-gram model consists of the words w_t inside the documents of a corpus C. For each word the output consist of context of w_t, the set $Context(w_t)$. The words in the input as well as the output are projected on an embedding. The specific feature of the word2vec approach is that two different embeddings are use for the words in the input and words in the output. Note that the denominator contains a term for every word in the vocabulary. With a vocabulary often larger than 100k words the associated computational effort is prohibitive. Therefore Mikolov et al. [2013] propose two important simplifications.

2.4.1 Skip-gram model with Hierarchical Softmax

Gradient Calculation

For Skip-gram model has to predict its context $Context(w)$ of the words using the current word w as input. Mikolov et al. [2013] propose the following the objective function

$$p(Context(w)|w) = \prod_{u \in Context(w)} p(u|w) \tag{2.5}$$

[25] COMMENT: What is the task of each model? Please specify at the beginning.
[26] COMMENT: C was defined without mathcal
[27] COMMENT: No proper sentence!!!

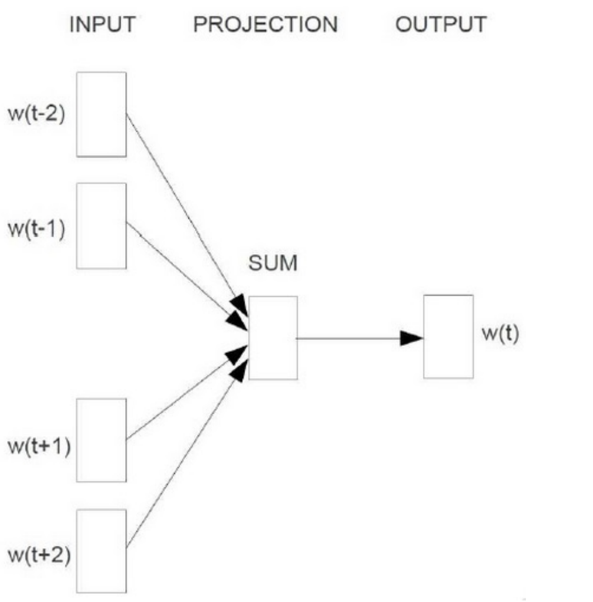


Figure 2.6: Structure of the CBOW model predicting the current word from its context words.

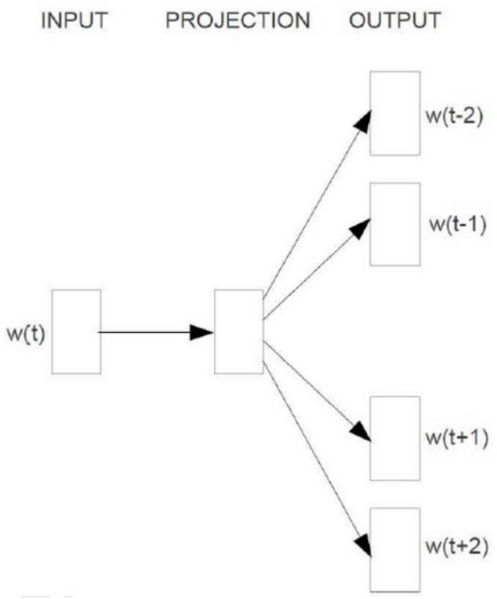


Figure 2.7: Structure of the Skip-Gram model predicting the context words from the current word.

$p(u|w)$ is the probability of a single output word u given the current word w.
[28] In the above formula $p(u|w)$ in accordance with section describes the Hierarchical Softmax thought, similar to (4.3) written as

$$p(u|w) = \prod_{j=2}^{l^u} p(d_j^u|\mathrm{v}(w), \theta_{j-1}^u),$$

among them

$$p(d_j^u|\mathbf{v}(w), \theta_{j-1}^u) = [\theta(\mathbf{v}(w)^{\mathrm{T}}\theta_{j-1}^u)]^{1-d_j^u} \cdot [1 - \theta(\mathbf{v}(w)^{\mathrm{T}}\theta_{j-1}^u)]^{1-d_j^u} \tag{2.6}$$

The (4.6) followed by generations back, you can get the log-likelihood function (4.2) of the specific expression

$$\begin{aligned}\mathcal{L} &= \sum_{w\in\mathcal{C}} \log \prod_{u\in Context(w)} \prod_{j=2}^{l^u} \{[\theta(\mathbf{v}(w)^{\mathrm{T}}\theta_{j-1}^u)]^{1-d_j^u} \cdot [1 - \theta(\mathbf{v}(w)^{\mathrm{T}}\theta_{j-1}^u)]^{\,d_j^u}\} \\ &= \sum_{w\in\mathcal{C}} \sum_{u\in Context(w)} \sum_{j=2}^{l^u} \{(1 - d_j^u) \cdot \log[\theta(\mathbf{v}(w)^{\mathrm{T}}\theta_{j-1}^u)] + d_j^u \cdot \log[1 - \theta(\mathbf{v}(w)^{\mathrm{T}}\theta_{j-1}^u)]\}. \end{aligned} \tag{2.7}$$

[28] COMMENT: I do not understand the following text!!!!! You have to motivate this by some hierarchical splitting example Huffman tree

Similarly, as in the following gradients of convenience, under the triple summation symbol braces contents of abbreviated as $\mathcal{L}(w, u, j)$, ie

$$\mathcal{L}(w,u,j) = (1 - d_j^u) \cdot \log[\theta(\mathbf{v}(w)^{\mathrm{T}}\theta_{j-1}^u)] + d_j^u \cdot \log[1 - \theta(\mathbf{v}(w)^{\mathrm{T}}\theta_{j-1}^u].$$

So far, it has been deduced logarithmic likelihood function of expressions like (4.7), which is the objective function Skip-gram model. Then also use stochastic gradient ascent method to optimize the key is to give two types of gradients. First consider $\mathcal{L}(w, u, j)$ on θ_{j-1}^u gradient calculation (with the corresponding portion of the model is derived CBOW completely analogous).

$$\partial\frac{\mathcal{L}(w,u,j)}{\partial\theta_{j-1}^u} = \frac{\partial}{\partial\theta_{j-1}^u}\{(1 - d_j^u) \cdot \log[\theta(\mathbf{v}(w)^{\mathrm{T}}\theta_{j-1}^u)] + d_j^u \cdot \log[\theta(\mathbf{v}(w)^{\mathrm{T}}\theta_{j-1}^u)]\}$$

[29]

2.4.2 Skip-gram with Negative Sampling

Negative Sampling (NEG) is proposed by Mikolov et al. [2013], which is the simplified version of NCE(Noise Contrastive Estimation)[30]. The purpose is to improve the training and the quality of word vectors. In comparison with Hierarchical Softmax, NEG do not use the Huffman tree. Instead, it use Random Negative Sampling, which can improve the performance much.[31]

The details of NCE is a little complex, the essence is to use a known probability density function to estimate an unknown probability density function. In short, assume there is an unknown probability density function Y and a known probability density function X, if we get the relationship between X and Y, we can obtain X as well.The detail of method reference to [NCE].

The objective function is:

$$G = \prod_{w \in \mathcal{C}} \prod_{u \in Context(w)} g(u), \tag{2.8}$$

Here, we want to maximize $\prod_{u \in Context(w)} g(u)$ giving $(w, Context(w)))$, and $g(u)$ is defined as

$$g(u) = p(z|w) * \prod_{z \in NEG(u)} p(z|w),$$

where $NEG(u)$ represents the negative samples[32] generated by u, the conditional probability[33]

$$p(z|w) = \sigma(\mathbf{v}(w)^{\mathrm{T}}\theta^z)$$

[29] COMMENT: Is this approach exact or an approximation?

[30] COMMENT: need a reference

[31] COMMENT: Where is this prooved?

[32] COMMENT: what is that?

[33] COMMENT: the negative samples are not generated by this conditional probability. What is θ?

It can also be written as one expression

$$p(z|w) = [\sigma(\mathbf{v}(w)^{\mathrm{T}}\theta^z)]^{L^u(z)} \cdot [1-\sigma(\mathbf{v}(w)^{\mathrm{T}}\theta^z)]^{1-L^u(z)} \tag{2.9}$$

where [34]

$$L^u(z) = \begin{cases} 1, & u = z; \\ 0, & u \neq z, \end{cases}$$

And then we use the log of G, so the final objective function is

$$\begin{aligned} L = \log\ G = \log \prod_{w\in\mathcal{C}} \prod_{u\in Context(w)} g(u) &= \sum_{w\in\mathcal{C}} \sum_{u\in Context(w)} \log\ g(u) \\ = \sum_{w\in\mathcal{C}} \sum_{u\in Context(w)} \log\left(p(u|w) \prod_{z\in NEG(u)} p(z|w)\right) \\ = \sum_{w\in\mathcal{C}} \sum_{u\in Context(w)} \left(\log p(u|w) + \sum_{z\in NEG(u)} \log\ p(z|w)\right) \\ = \sum_{w\in\mathcal{C}} \sum_{u\in Context(w)} \sum_{z\in\{u\}\cup NEG(u)} \log\ \{[\sigma(\mathbf{v}(w)^{\mathrm{T}}\theta^z)]^{L^u(z)} \cdot [1-\sigma(\mathbf{v}(w)^{\mathrm{T}}\theta^z)]^{1-L^u(z)}\} \\ = \sum_{w\in\mathcal{C}} \sum_{u\in Context(w)} \sum_{z\in\{u\}\cup NEG(u)} \{L^u(z)\cdot\log[\sigma(\mathbf{v}(w)^{\mathrm{T}}\theta^z)] + [1-L^u(z)]\cdot\log[1-\sigma(\mathbf{v}(w)^{\mathrm{T}}\theta^z)]\}. \end{aligned}$$

In order to calculate the gradient more conveniently, we use $L(w,u,z)$ to represent the contents of curly braces as

$$\mathcal{L}(w,u,z) = L^u(z)\cdot\log[\sigma(\mathbf{v}(w)^{\mathrm{T}}\theta^z)] + [1-L^u(z)]\cdot\log[1-\sigma(\mathbf{v}(w)^{\mathrm{T}}\theta^z)]$$

Mikolov et al. [2013] use the **Stochastic gradient ascent method** to minimize $\mathcal{L}(w,u,z)$. The point is to calculate two kinds of gradient. Let's consider the gradient with respect to θ^z firstly.

$$\begin{aligned} &\frac{\partial\mathcal{L}(w,u,z)}{\partial\theta^z} \\ &= \frac{\partial}{\partial\theta^z}\{L^u(z)\cdot\log[\sigma(\mathbf{v}(w)^{\mathrm{T}}\theta^z)] + [1-L^u(z)]\cdot\log[1-\sigma(\mathbf{v}(w)^{\mathrm{T}}\theta^z)]\} \\ &= L^u(z)[1-\sigma(\mathbf{v}(w)^{\mathrm{T}}\theta^z)]\mathbf{v}(w) - [1-L^u(z)]\sigma(\mathbf{v}(w)^{\mathrm{T}}\theta^z)\mathbf{v}(w) \\ &= \{L^u(z)[1-\sigma(\mathbf{v}(w)^{\mathrm{T}}\theta^z)] - [1-L^u(z)]\sigma(\mathbf{v}(w)^{\mathrm{T}}\theta^z)\}\mathbf{v}(w) \\ &= [L^u(z)-\sigma(\mathbf{v}(w)^{\mathrm{T}}\theta^z)]\mathbf{v}(w). \end{aligned}$$

Thus, the updating formula of θ^z can be written as

$$\theta^z := \theta^z + \eta[L^u(z)-\sigma(\mathbf{v}(w)^{\mathrm{T}}\theta^z)]\mathbf{v}(w).$$

[34] COMMENT: L was the likelihood, choose another letter

The next, let's consider the gradient of $\mathbf{v}(w)$. Using the symmetry of $\mathbf{v}$(w) and θ^z, we have

$$\frac{\partial \mathcal{L}(w,u,z)}{\partial \mathbf{v}(w)} = [L^u(z) - \sigma(\mathbf{v}(w)^{\mathrm{T}}\theta^z)]\theta^z,$$

Thus, the updating formula of $\mathbf{v}(u)$ can be written as

$$\mathbf{v}(w) := \mathbf{v}(w) + \eta \sum_{z \in \{u\} \cup NEG\{u\}} \frac{\partial \mathcal{L}(w,u,z)}{\partial \mathbf{v}(w)}$$

$$= \mathbf{v}(w) + \eta \sum_{z \in \{u\} \cup NEG\{u\}} [L^u(z) - \sigma(\mathbf{v}(w)^{\mathrm{T}}\theta^z)]\theta^z.$$

2.5 Huang's Model

The work of Huang et al. [2012] is based on the model of Collobert and Weston [2008]. They try to make embedding vectors with richer semantic information. They had two major innovations to accomplish this goal : The first innovation is using global information from the whole text to assist local information, the second innovation is using the multiple word vectors to represent polysemy.
Huang thinks Collobert and Weston [2008] use only "local context". In the process of training vectors, they used only 10 words as the context for each word, counting the center word itself, there are totally 11 words' information. This local information can not fully exploit the semantic information of the center word. Huang used their neural network directly to compute a score as the "local score".
And then Huang proposed a "global information", which is somewhat similar to the traditional bag of words model. Bag of words is about accumulating One-hot Representation from all the words of the article together to form a vector (like all the words thrown in a bag), which is used to represent the article. Huang's global information used the average weighted vectors from all words in the article (weight is word's idf), which is considered the semantic of the article. He connected such semantic vector of the article (global information) with the current word's vector (local information) to form a new vector with double size as an input, and then used the C&W's network to calculate the score. Figure [huang] shows such structure. With the "local score" from original C&W approach and "Global score" from improving method based on the C&W approach, Huang directly add two scores as the final score. The final score would be optimized by the pair-wise target function from C&W. Huang found his model can capture better semantic information.

The second contribution of this paper is to represent polysemy using multiple embeddings for a single word. [?] also mentioned that this would be an very important issue, but he was still looking for a solution, now Huang gives an idea. For each center word, he took 10 nearest context words and calculated the weighted average of the embeddings of these 10 word vectors (idf weights) as the context vector. Huang used all context vectors

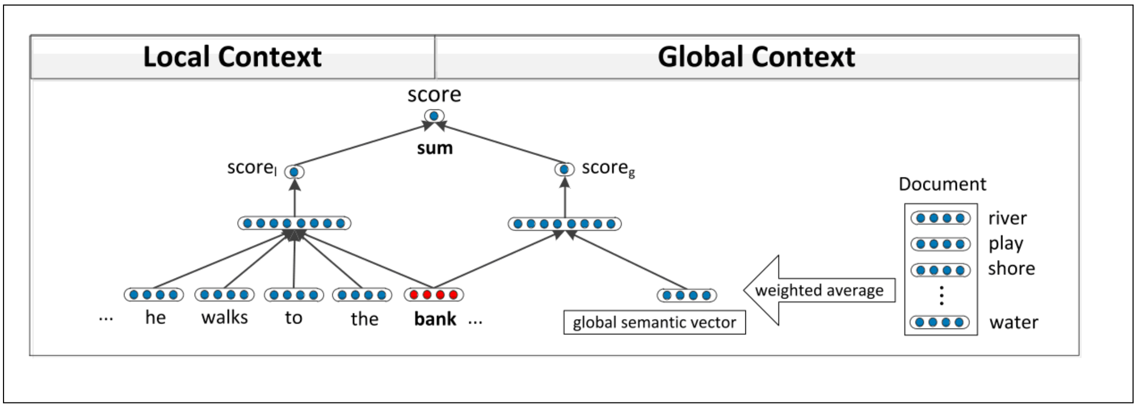


Figure 2.8: The network structure from [?]

Table 2.1: Senses computed with Huang's network and their nearest neighbors.

Center Word	Nearest Neighbors
bank_1	corporation, insurance, company
bank_2	shore, coast, direction
star_1	movie, film, radio
star_2	galaxy, planet, moon
cell_1	telephone, smart, phone
cell_2	pathology, molecular, physiology
left_1	close, leave, live
left_2	top, round, right

to do a k-means clustering. The number of clusters [35]. He relabel each word based on the clustering results (different classes of the same words would be considered as different words to process)[36]. Finally he re-trained the word vectors. Table 2.1 gives some examples from his model's results.

2.6 EM-Algorithm based method

Tian et al. [2014] proposed an approach based on the EM-algorithm from . This method is the extension of the normal skip-gram model. They still use each center word to predict several context words. The difference is that each center word can have several senses with different probabilities. The probability should represent the relative frequency of the sense in the corpus. For example, considering $bank_1$ in the sense of "side of the river" and

[35] COMMENT: where does he get them??
[36] COMMENT: How was this exactly done?

Table 2.2: Word senses computed by Tian et al.

word	Prior Probability	Most Similar Words
apple_1	0.82	strawberry, cherry, blueberry
apple_2	0.17	iphone, macintosh, microsoft
bank_1	0.15	river, canal, waterway
bank_2	0.6	citibank , jpmorgan, bancorp
bank_3	0.25	stock, exchange, banking
cell_1	0.09	phones cellphones, mobile
cell_2	0.81	protein, tissues, lysis
cell_3	0.01	locked , escape , handcuffed

$bank_2$ meaning "financial institution". Usually $bank_1$ will have a smaller probability and $bank_2$. We can say in the corpus, in most sentences of the corpus the word "bank" means "financial institution" and in other fewer cases it means "side of the river".

Objective Function

Considering w_I as the input word and w_O as the output word, (w_I, w_O) is a data sample. The input word w_I have N_{w_I} prototypes, and it appears in its h_{w_I}-th prototype, i.e., $h_{w_I} \in \{1, .., N_{w_I}\}$ [] The prediction $P(w_O|w_I)$ is like the following formula

$$p(w_O|w_I) = \sum_{i=1}^{N_{w_I}} P(w_O|h_{w_I} = i, w_I) P(h_{w_I} = i|w_I) = \sum_{i=1}^{N_{w_I}} \frac{exp(U_{w_O}^{\mathrm{T}} V_{w_I,i})}{\sum_{w \in W} exp(U_w^{\mathrm{T}} V_{w_I,i})} P(h_{w_I} = i|w_I)$$

where $V_{w_I,i} \in R^d$ refers to the d-dimensional "input" embedding vector of w_I's i-th prototype and $U_{w_O} \in R^d$ represents the "output" embedding vectors of w_O. Specifically, they use the Hierarchical Softmax Tree function to approximate the probability calculation.

Algorithm Description

Particularly for the input word w, they put all samples (w as the input word) together like $\{(w, w_1), (w, w_2), (w, w_3)...(w, w_n)\}$ as a group. Each group is based on the input word. So the whole training set can be separated as several groups. For the group mentioned above, one can assume the input word w has m vectors (m senses), each with the probability $p_j (1 \leq j \leq m)$. And each output word $w_i (1 \leq i \leq n)$ has only one vector.

In the training process, for each iteration, they fetch only part of the whole training set and then split it into several groups based on the input word. In each E-step, for the group mentioned above, they used soft label $y_{i,j}$ to represent the probability of input word in sample (w, w_i) assigned to the j-th sense. The calculating of $y_{i,j}$ is based on the value of sense probability and sense vectors. After calculating each $y_{i,j}$ in each data group, in the M-step, they use $y_{i,j}$ to update sense probabilities and sense vectors from input word, and the word vectors from output word. Table 2.2 lists some results from this model.

2.7 A Method to Determine the Number of Senses

Neelakantan et al. [2015] [37]

[37] COMMENT: Write a paragraph on this

Chapter 3

Solution

Generally speaking, our model is a extension of skip-gram model with negative sampling. We assume each word in the sentence can have one or several senses. But unlike Huang's model (?), they use clustering results to label word senses and once assigned these senses can not be changed. Our model is different, we do not use any pre work to assign senses (label words), instead we just assign each word with random senses and they can be adjusted afterwards. We also follow the idea from EM-Algorithm based method ([?]), word's different senses have different probabilities, the probability can represent if a sense is used frequent in the corpus.

In fact, after some experiments, we found our original model is not good. So we simplified our original model. Anyhow we will introduce our original model and show the failures in the next chapter, and explain the simplification.

3.1 Definition

C is the corpus and D is the vocabulary. We consider that the training corpus C is made up by M sentences, like $(S_1, S_2, \ldots, S_M)$, and each sentence is made up by several words like $S_i = (w_{i,1}, w_{i,2}, \ldots, w_{i,L_i})$ and L_i is the length of sentence S_i. We use $w_{i,j}$ to represent the word token in the position j of sentence S_i. Each word in each sentence has one or multiple senses. We use h to lookup table of sense assignment, specifically $h_{i,j}$ is the sense index of word $w_{i,j}$ $(1 \leq h_{i,j} \leq N_{w_{i,j}})$, where N_w is the max number of senses of word w $(w \in D)$

We use V and U to represent respectively the set of input embedding vectors and the set of output embedding vectors respectively. And each embedding vectors has the dimension d. Additionally, $V_{w,s} \in \mathbb{R}^d$ means the input embedding vectors from sense s of word w, similarly as the definition of $U_{w,s}$, where $w \in D$, $1 \leq s \leq N_w$. Following the Skip-gram model with negative sampling, K is the number of negative samples and c is the size of context. And $P_n(w)$ is the smoothed unigram distribution which is used to generate negative samples. Specifically, $P_n(w) = \frac{count(w)^{\frac{3}{4}}}{(\sum_{i=1}^{M} L_i)^{\frac{3}{4}}}$ $(w \in D)$, where $count(w)$ is the number of times w occurred in C and $\sum_{i=1}^{M} L_i$ is the number of total words in C.

3.2 Objective Function

$$\begin{aligned} G = \frac{1}{M}\sum_{i=1}^{M}\frac{1}{L_i}\sum_{t=1}^{L_i}\sum_{\substack{-c \le j \le c \\ j \neq 0 \\ 1 \le j+t \le L_i}} & \Bigg(\log p\Big[(w_{i,t+j}, h_{i,t+j})|(w_{i,t}, h_{i,t})\Big] \\ & + \sum_{k=1}^{K}\mathbb{E}_{z_k \sim P_n(w)}\log\Big\{1 - p\Big[[z_k, R(N_{z_k})]|(w_{i,t}, h_{i,t})\Big]\Big\}\Bigg) \end{aligned} \tag{3.1}$$

where $p\Big[(w', s')|(w, s)\Big] = \sigma({U_{w',s'}}^{\mathrm{T}} V_{w,s})$ and $\sigma(x) = \frac{1}{1+\mathrm{e}^{-x}}$.

$p\Big[(w_{i,t+j}, h_{i,t+j})|(w_{i,t}, h_{i,t})\Big]$ is the probability of using center word $w_{i,t}$ with sense $h_{i,t}$ to predict one surrounding word $w_{i,t+j}$ with sense $h_{i,t+j}$, which needs to be **maximized**. $[z_1, R(N_{z_1})]$,…,$[(z_K, R(N_{z_K})]$ are the negative sample words with random assigned senses to replace $(w_{i,t+j}, h_{i,t+j})$, and $p\Big[[z_k, R(N_{z_k})]|(w_{i,t}, h_{i,t})\Big]$ $(1 \le k \le K)$ is the probability of using center word $w_{i,t}$ with sense $h_{i,t}$ to predict one negative sample word z_k with sense $R(N_{z_k})$, which needs to be **minimized**. It is noteworthy that, $h_{i,t}$ ($w_{i,t}$'s sense) and $h_{i,t+j}$ ($w_{i,t+j}$'s sense) are assigned advance and $h_{i,t}$ may be changed in the **Assign**. But z_k's sense s_k is always assigned randomly.

The final objective is to find out optimized parameters $\theta = \{h, U, V\}$ to maximize the Objective Function G, where h is updated in the **Assign** and $\{U, V\}$ is updated in the **Learn**.

When the center word $w_{i,t}$ is giving, we use **score function** $f_{i,t}$ with fixed negative samples $\bigcup_{\substack{-c \le j \le c \\ j \neq 0 \\ 1 \le j+t \le L_i}} [(z_{j,1}, s_{j,1}), \ldots, (z_{j,K}, s_{j,K})]$ (senses are assigned randomly already)

$$f_{i,t}(s) = \sum_{\substack{-c \le j \le c \\ j \neq 0 \\ 1 \le t+j \le L_i}} \Bigg(\log p[(w_{i,t+j}, h_{i,t+j})|(w_{i,t}, s)] + \sum_{k=1}^{K}\log\Big\{1 - p[(z_{j,k}, s_{j,k})|(w_{i,t}, s)]\Big\}\Bigg)$$

to select the "best" sense (with the max value) of each center word in the **Assign**.

Taking $[(w_{i,t}, h_{i,t}), (w_{i,t+j}, h_{i,t+j})]$ as a training sample, we define **loss function** *loss* for each sample as

$$loss\big((w_{i,t}, h_{i,t}), (w_{i,t+j}, h_{i,t+j})\big)$$

$$= -\log\, p\Big[(w_{i,t+j}, h_{i,t+j})|(w_{i,t}, h_{i,t})\Big] - \sum_{k=1}^{K} \mathbb{E}_{z_k \sim P_n(w)} \log\, \Big\{1 - p\Big[[z_k, R(N_{z_k})]|(w_{i,t}, h_{i,t})\Big]\Big\}$$

Here the loss is defined as the negative log probability.

And the loss function of whole corpus is

$$loss(C) = \frac{1}{M}\sum_{i=1}^{M}\frac{1}{L_i}\sum_{t=1}^{L_i}\sum_{\substack{-c\,\le\, j\,\le\, c\\ j\,\ne\, 0\\ 1\,\le\, j+t\,\le\, L_i}} loss\big((w_{i,t}, h_{i,t}), (w_{i,t+j}, h_{i,t+j})\big)$$

After **Assign**, h is fixed. So we the same method in the normal Skip-gram with negative sampling model (stochastic gradient decent) to minimize G in the **Learn**. So the objective of **Learn** is

$$\arg\min_{\{V,U\}} \frac{1}{M}\sum_{i=1}^{M}\frac{1}{L_i}\sum_{t=1}^{L_i}\sum_{\substack{-c\,\le\, j\,\le\, c\\ j\,\ne\, 0\\ 1\,\le\, j+t\,\le\, L_i}} loss\big((w_{i,t}, h_{i,t}), (w_{i,t+j}, h_{i,t+j})\big)$$

Use

$$N = \frac{1}{M}\sum_{i=1}^{M}\frac{1}{L_i}\sum_{t=1}^{L_i}\sum_{\substack{-c\,\le\, j\,\le\, c\\ j\,\ne\, 0\\ 1\,\le\, j+t\,\le\, L_i}} 1$$

to represent the number of total training samples in one epoch. (An epoch is a measure of the number of times all of the training samples are used once.) .

Use stochastic gradient descent:

- For N Iterations:
  - For each training sample $(w_{i,t}, w_{i,t+j})$
    - Generate negative sample words to replace $w_{i,t+j}$: $(w_1, \dots, z_k)$
    - Calculate the gradient $\Delta = -\nabla_{\{V,U\}} loss(w_{i,t}, w_{i,t+j})$
    - Δ is only made up by $\{\Delta_{V_{w_{i,t}}}, \Delta_{U_{w_{i,t+j}}}, [\Delta_{U_{w_1}}, \dots, \Delta_{U_{z_k}}]\}$
    - Update Embeddings:
      - $V_{w_{i,t}} = V_{w_{i,t}} + \alpha\Delta_{V_{w_{i,t}}}$
      - $U_{w_{i,t+j}} = U_{w_{i,t+j}} + \alpha\Delta_{U_{w_{i,t+j}}}$
      - $U_{z_k} = U_{z_k} + \alpha\Delta_{U_{z_k}}, 1 \le k \le K$

    (α is the learning rate and will be updated every several iterations)

3.3 Alrogithm Description

In the beginning, in each word of each sentence, senses are assigned **randomly**, that is $h_{i,j}$ is set to any value between 1 to $N_{w_{i,j}}$. $N_{w_{i,j}}$ can be decide by the count of word in corpus. If the count is much, the max number of senses would be much as well. Every sense have both input embedding and output embedding, although the final experiment results shows that output embedding should have only one sense.

The training algorithm is an iterating between **Assign** and **Learn**. The **Assign** is to use the **score function** (sum of log probability) to select the best sense of the center word. And it uses above process to adjust senses of whole sentence and repeats that until sense assignment of the sentence is stable (not changed). The **Learn** is to use the new sense assignment of each sentence and the gradient of the **loss function** to update the input embedding and output embedding of each sense (using stochastic gradient decent).

Initialization
Input embedding vectors and output embedding vectors will be initialized from the normal Skip-gram model, which can be some public trained word vectors dataset. But in the next chapter, our experiment actually always do two steps. The first step is like normal skip-gram model and all words have only one sense. After that , the second step will use the result from that to initialize . Specifically, we use word embedding vectors from normal skip-gram model pluses some small random value (vector) to be their sense embedding vectors. Of course for different senses of the same word, the random values (vectors) are different. So in the beginning, sense vectors of each word are different but similar.

Sense Probabilities
Each word has several senses. Each sense has a probability, in initialization they are set equally. For each assignment part, the probability will change based on the number of selected. Notice that , EM-Algorithm also uses sense probabilities. But our purpose to use sense probability is different. In their model, each frequent word has several senses in the meantime with different probabilities, and in each iteration they will update the probabilities and all sense embedding vectors. While in our model, in each iteration, each word can only have one sense which can be adjusted, and after **Assign**, we only update the assigned sense. But we still use sense probabilities. The usefulness is also about recording the sense frequency, that is the assigned frequency. Some senses are selected in the **Assign**, their relative probabilities will increase. Correspondingly, for other senses which are not selected, their probabilities will decrease.
So what is useful of these sense probabilities? Actually, they are not just used to record the assigned frequency. If some sense's probability is too low, we will use some frequent sense (assigned frequently) to reset this sense with some small random value (vector) as the same operation in the initialization. Otherwise, the infrequent assigned senses in the early iterations will always be ignored in the next iterations. Actually, we already did some

experiments without sense probabilities and these experiments' results really told use the above situation.

Next, we will describe the specific steps of **Assign** and **Learn** in the form of pseudo-code.

Assign:

```
FOR $i$:= 1 TO $M$
    DO
        FOR $t$:= 1 TO $L_i$
            $h_{i,t} = \max_{1 \leq s \leq N_{w_{i,t}}} f_{i,t}(s)$
        END
    UNTIL no $h_{i,t}$ changed
END
```

Learn:

```
FOR $i$:= 1 TO $M$
    FOR $t$:= 1 TO $L_i$
        FOR $j$:= $-c$ TO $c$
            IF $j \neq 0$ AND $t + j \geq 1$ AND $t + j \leq L_i$ THEN

                generate negative samples $\big[(z_1, s_1), \ldots, (z_K, s_K)\big]$

                $\Delta = -\nabla_\theta loss\big((w_{i,t}, h_{i,t}), (w_{i,t+j}, h_{i,t+j})\big)$
                $\Delta$ is made up by $\{\Delta_{V_{w_{i,t},h_{i,t}}}, \Delta_{U_{w_{i,t+j},h_{i,t+j}}}, [\Delta_{U_{w_1,w_1}}, \ldots, \Delta_{U_{z_k,z_k}}]\}$

                $V_{w_{i,t},h_{i,t}} = V_{w_{i,t},h_{i,t}} + \alpha\Delta_{V_{w_{i,t},h_{i,t}}}$
                $U_{w_{i,t+j},h_{i,t+j}} = U_{w_{i,t+j},h_{i,t+j}} + \alpha\Delta_{U_{w_{i,t+j},h_{i,t+j}}}$
                $U_{z_k,s_k} = U_{z_k,s_k} + \alpha\Delta_{U_{z_k,s_k}}, 1 \leq k \leq K$

            END
        END
    END
END
```

The detail of gradient calculation of $loss\big((w_{i,t}, h_{i,t}), (w_{i,t+j}, h_{i,t+j})\big)$ is

$$\Delta_{V_{w_{i,t},h_{i,t}}} = -\frac{\partial loss\big((w_{i,t}, h_{i,t}), (w_{i,t+j}, h_{i,t+j})\big)}{\partial V_{w_{i,t},h_{i,t}}}$$

$$= [1 - \log\, \sigma(U_{w_{i,t+j},h_{i,t+j}}{}^{\mathrm{T}} V_{w_{i,t},h_{i,t}})] U_{w_{i,t+j},h_{i,t+j}} + \sum_{k=1}^{K} [-\log\, \sigma(U_{z_k,s_k}{}^{\mathrm{T}} V_{w_{i,t},h_{i,t}}))] U_{z_k,s_k}$$

$$\begin{aligned}\Delta_{U_{w_{i,t+j},h_{i,t+j}}} &= -\frac{\partial loss\big((w_{i,t}, h_{i,t}), (w_{i,t+j}, h_{i,t+j})\big)}{\partial U_{w_{i,t+j},h_{i,t+j}}} \\ &= [1 - \log\, \sigma(U_{w_{i,t+j},h_{i,t+j}}{}^{\mathrm{T}} V_{w_{i,t},h_{i,t}})] V_{w_{i,t},h_{i,t}}\end{aligned}$$

$$\begin{aligned}\Delta_{U_{z_k,s_k}} &= -\frac{\partial loss\big((w_{i,t}, h_{i,t}), (w_{i,t+j}, h_{i,t+j})\big)}{\partial U_{z_k,s_k}} \\ &= [-\log\, \sigma(U_{z_k,s_k}{}^{\mathrm{T}} V_{w_{i,t},h_{i,t}}))] V_{w_{i,t},h_{i,t}}\end{aligned}$$

Iterating between **Assign** and **Learn** till the convergence of the value of G makes the whole algorithm complete. Actually, we use the loss of validation set to monitor if the training process is convergence. After a couple of iterations, we do the similar **Assign** operation on validation set and then calculate the loss. To be noted that, the **Assign** on validation set is a little different from the one on training set. Here, the negative samples needs to be always fixed throughout the training process. Another thing is that validation set and training set should not be overlapped. As long as the validation loss begin to increase. We stop training. And select the result with best validation loss as the final result.

Chapter 4

Implementation

For the implementation, we use spark framework to implement our model. In this chapter, we will firstly introduce some knowledge about spark and how we use these techniques to implement our model. After that, we will introduce the experiments we did and analysis our results.

4.1 Introduction of Spark

Spark has one driver and several executors. Usually, an executor is a cpu core, and we call each machine as worker, so each worker has several executors. But logically we only need the driver and the executors, only for something about tuning we should care about the worker stuff, e.g. some operations need to do communication between different machine. But for most of cases, each executor just fetch part of data and deal with it, and then the driver collect data from all executors.

Spark has many useful things. What I use are about RDD (Resilient Distributed Datasets) and some operations on RDD, which is a special data structure containing the data set. and is convenient to be These operations have two type, one is Transformation operation, another is action operation. Firstly, Spark reads text file from file system (normally it is HDFS). And now, the data is the form of RDD. The transformation operation is to transform a RDD to another RDD. RDD has two types, the one read from file system is called HDFS RDD, the one transformed by transformation operation is called map RDD. Generally after some transformation operations, people use the action operation to gain some useful information from the last RDD. To be noted that, transformation is not to change the element in the current RDD, insead it create a new RDD. How about the old one? If you really need it, you can store it in catch, memory or the disk. Some times we store it (catch it) not only for the intermediate results, but also for some algorithms required iteration. For instance, if you want to use action operation to gain some Transformation operation is mainly about map and filter, which is very similar as the operations in any functional programming. And the action operation is mainly about aggregate, reduce, count, and collect. RDD can be operated only by transformation operation and stored logically in every executor. Data in RDD can not be changed. Spark use some operations to Each transformation operation will create a new RDD and would not change the DATA in the original RDD.

4.2 Implementation

We use $syn0$ to represent the input embedding V and $syn1$ to represent the output embedding U. $syn0$ and $syn1$ are set to be as the broadcast variables, which is only readable and can not be changed by executors.

Assign Step
In the assignment, we use map transformation to transform each sentence with senses information to another sentence with changed senses information. So one RDD becomes to another RDD. In this process, $syn0$ and $syn1$ will be used (only read) to calculate the loss.

Learn Step
In the training, we also use map transformation. Instead of transforming sentences to sentences, we transform the original sentence RDD into the collection of updated $syn0$ and updated $syn1$. Yes, we update $syn0$ and $syn1$ in this process, because we need to train our parameters. But executors can really change the $syn0$ and $syn1$ directly. So we copy these broadcast variables to the local $syn0$ and $syn1$ in each executor, so that each executor has its own $syn0$ and $syn1$ and update them independently. And then we use the average of them as the new global $syn0$ and $syn1$.

So each executor has two vectors (representing $syn0$ and $syn1$ respectively). And then we use *treeAggregate* to collect all such vectors together from different executor (cpu core). In the aggregation, different $syn0$'s vectors add up together, and different $syn0$'s vectors add up together. Finally, we get one $syn0$ and one $syn1$. For now, we set them as new $syn0$ and $syn1$, which will be used as the broadcast value in the next iteration.

After getting the new global $syn0$ and $syn1$, because they are added up by several ones, some values of some embeddings may be very big. Thus, we need to do normalization to avoid to big values. Our normalization method is very simple, which is to check all embeddings from $syn0$ and $syn1$ if the length is bigger than 4, if that we just normalization them to the new embeddings with length of 4.

4.3 Experiment

Data preparing
We use the same corpus as other papers used, a snapshot of Wikipedia at April, 2010 (Shaoul, 2010), which has 990 million tokens. Firstly we count the all words in the corpus. We transform all words to lower capital and then generate our vocabulary (dictionary). Actually, we set a *minCount*, if the word count is smaller than this value. We remove it from corpus, so it won't be in the vocabulary. And then we calculate the frequency of word count. For example, there are 300 words which has count 10. So the frequency of count 10 is 300. After that, we can calculate the accumulated frequency, which can help us to choose the *minCount*. That is, if accumulated frequency of count 200 is 100000, there would be 100000 words whose count is at least 200. So we can adjust the different accumulated frequency to get different vocabulary size.

Environment
Our program is running on a single machine with 32 cores. For some experiment, we use all cores as executers. We also tried some experiments on several machine, but that is not very good for our program, we will explain some reasons later. So there is no communication between different machines. But there are some experiments requiring fewer cores.

Training set and validation set
We split corpus as training set and validation set. Training set has 99% data and validation set has only 1% data. We use validation set to monitor our training process if it is convergence. If an training algorithm is convergence, the loss of validation set should be at the lowest value. And then it will gradually increase, which means the training is over-fitting. So we will calculate the loss of validation set every several training iterations and then compare with the previous validation loss, if the current value is bigger than previous value, we stop our training process and fetch the previous result as the final result to store into the disk. That is, after each calculating the loss of validation set, we will store our results, and we won't do anything in the step to stop.

To be noted that, because we want to use the validation set to monitor our training, so the validation set and training set should not be overlapping. And another import thing is that, the negative samplings of validation set should always be fixed. The assignment step for validation set is almost same as the one for training set. The only different thing is that, the negative samples for each word of each sentence in the validation set is not changed. But for each iteration of assignment for sentence in training set, the negative sampling are new.

Actually, for our cases, it is not very easy to choose the vocabulary size and the vector size. For each experiment, we record the *count* time and *treeAggregate* time. It is also difficult to choose the number of RDD, which although is not the very important factor to decide the final results. We found that the number of RDDs only affect the time and space using, it would not affect the final loss.

We need two steps to train sense embedding vectors. The first step is to set the number of sense only one and train normal word embedding vectors. In second step, the program will use the result from the first step to do initialization and then train the sense embedding vectors.

We did 8 different experiments for the first step shown as 4.3, based on that we did another 11 experiments for the second step 4.4. For these two tables, we give another notation table as 4.1 to explain the meaning of each label. Considering other parameters like the number of RDDs and window size is always fixed. The The difference for parameters between step 1 and step 2 is that,

id	The id number of the experiment.
c1	Min count for involving in Vocabulary
vec	Vector size for each embedding vector
cm	Threshold array for different number of senses
lr	The learning rate at the beginning of the experiment.
gm	The reduction rate of learning rate for each iteration
syn1	Whether each word has only one output embedding vector
t1	The average Training time of each iteration (excluding the parameter collection.
t2	The average treeAggregate operation time of each iteration
iter	The number of training iterations
t3	The average time of each iteration (including Assign step, train step and parameters collection)
t4	Total training time
loss	The best loss of validation set
SCWS	The Spearman's rank correlations on SCWS dataset.
word353	The Spearman's rank correlations on word353 dataset
init	The id number of experiment in step 1 (used to initialize embedding vectors)

Table 4.1: Notation explaination for parameters evaluation

numRDD=20	The number of RDD to split training data set.
windowSize=5	The window size
numNegative=10	

Table 4.2: Fixed value

id	c1	vec	lr	gm
(1)	200	300	0.1	0.9
(2)	200	250	0.1	0.9
(3)	200	200	0.1	0.9
(4)	200	150	0.1	0.9
(5)	200	100	0.1	0.9
(6)	200	50	0.1	0.9
(7)	20	50	0.1	0.9
(8)	20	50	0.01	0.95

Table 4.3: 5 Different Experiments in Step 1

id	c1	vec	cm	lr	gm	syn1	init
1	200	300	2000_10000	0.1	0.9	true	(1)
2	200	250	2000_10000	0.1	0.9	true	(2)
3	200	200	2000_10000	0.1	0.9	true	(3)
4	200	150	2000_10000	0.1	0.9	true	(4)
5	200	100	2000_10000	0.1	0.9	true	(5)
6	200	50	2000_10000	0.1	0.9	true	(6)
7	20	50	2000_10000	0.1	0.9	true	(7)
8	20	50	2000_10000	0.01	0.95	true	(8)
9	20	50	2000_100000	0.1	0.9	true	(7)
10	20	50	7000_10000	0.1	0.9	true	(7)
11	20	50	2000_10000	0.1	0.9	false	(7)

Table 4.4: 9 Different Experiments in Step 2

Chapter 5

Evaluation

In the following analysis, we use three different methods to assistant to our works. Nearest Neighbours, Similarity Task including WordSim-353 [] Dataset and the Contextual Word Similarity (SCWS) dataset from Huang[], and t-SNE based Visualization.

WordSim-353 Dataset is a Dataset made up by 353 pairs of words following by similarity scores from 10 different people and an average score. SCWS Dataset has 2003 pairs with their context respectively, which also contains 10 scores from 10 different people and an average score. For the WordSim-353 dataset, we use the maxSim function to calculate the similarity score of two words from our model as following:

$$maxSim(w, w') = max\{Cosine(V_{w,i}, V_{w',j})\}, (1 \leq i \leq N_w, 1 \leq j \leq N_{w'})$$

where $Cosine(x, y)$ denotes the cosine similarity score of vector x and y. And as the same notation from last chapter, N_w means the number of senses for word w, and $V_{w,i}$ represents the i-th sense input embedding vectors of word w. For the SCWS dataset, it is similar as the **Assign** operation, we use center word to predict the context words. But here we do not do the real assignment for whole sentence which needs several times to assign until it is stable. Actually, our sense output embedding has only one sense. So we just use the normal skip-gram model's prediction function to select the best center word's sense.

5.1 Parameters Comparison

Different parameters and hyper parameter can generate different loss and spend different time and memory space. Totally we did 9 experiments as following. In the early version , we try many different parameters on the small dataset and found that the number of negative samples, the window size are not the typical factors to affect the final results. And we also found that it is better to choose $numRDDs = 20$, which can balance the *count* time and *treeAggregate* time. Unfortunately, we lost some data of these early experiments. So in this section we only compare the following different experiments.

In the following, we will fix some parameters and build 5 comparison groups based these 9 experiments to check how different parameters affect the final validation loss, the convergence speed, training time and similarity results. Table 4.4 show the different

Different vector size

From the comparison in Table 5.1 , it is clear that the vector size is not the key factor to affect the final loss, even though the loss from experiment 3 is a little better. And there is another interesting thing that, when vector size if bigger, the score from SCWS is better but the score from word353 is worse.

id	vec	t1	t2	iter	t3	t4	loss	SCWS	word353
1	300	947.8	842	35	2272.9	79550	0.2437	0.5048	0.5233
2	250	764.7	533	35	1755.7	61450	0.2437	0.5083	0.5271
3	200	632.5	322	40	1389.9	55593	0.2436	0.5103	0.5371
4	150	502.7	210	35	1069.9	37448	0.2440	0.5048	0.5233
5	100	494.7	70.1	35	827.30	28956	0.2446	0.4994	0.5355
6	50	342.9	34.6	35	683.29	23915	0.2458	0.4666	0.5449

Table 5.1: Different Vector Size Comparison

Different Min Count

We can find from Table 5.2 , the size of dictionary is not the important factor based on loss or similarity tasks. Min count is used to remove some words which is not frequent. As we know , each word's embedding vector is trained based on the surrounding words. Since those words are infrequent, each of them involve the training of frequent words very few. So they won't affect the final embedding vectors of frequent words.

Table 5.2: Different Min Count Comparison

id	c1	t1	t2	iter	t3	t4	loss	SCWS	word353
6	200	342.9	34.6	35	683.3	23915	0.2458	0.4666	0.5449
7	20	849.0	343	35	1838.1	64335	0.2457	0.4371	0.4891

Different Sense Count Comparison

From Table 5.3 tell us ...

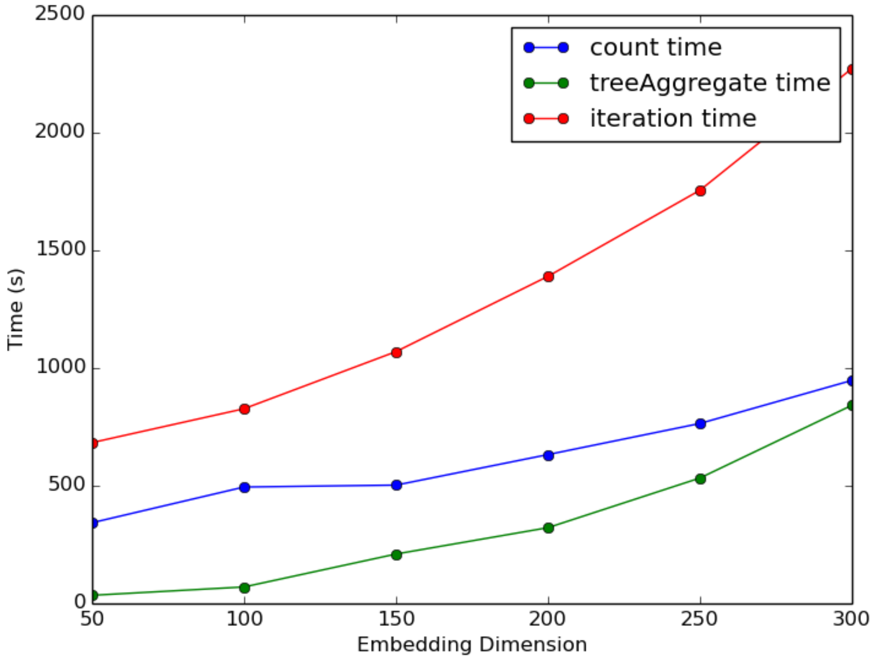


Figure 5.1: Shows the effect of varying embedding dimensionality of our Model on the Time

Table 5.3: Different Sense Count Comparison

id	cm	t1	t2	iter	t3	t4	loss	SCWS	word353
7	2000_10000	849	343	35	1838	64335	0.2457	0.4371	0.4891
9	2000_100000	798	338	35	1712	59912	0.2465	0.443	0.498
10	7000_10000	808	340	35	1740	60909	0.2462	0.4351	0.506

Different Learning Rate and Gamma
Table 5.4 shows that ...

id	lr	gm	t1	t2	iter	t3	t4	loss	SCWS	word353
7	0.1	0.9	849	343	35	1838	64335	0.246	0.4371	0.4891
8	0.01	0.95	797	370	40	1851	74032	0.267	..	..

Table 5.4: Different Learning Rate and Gamma Comparison

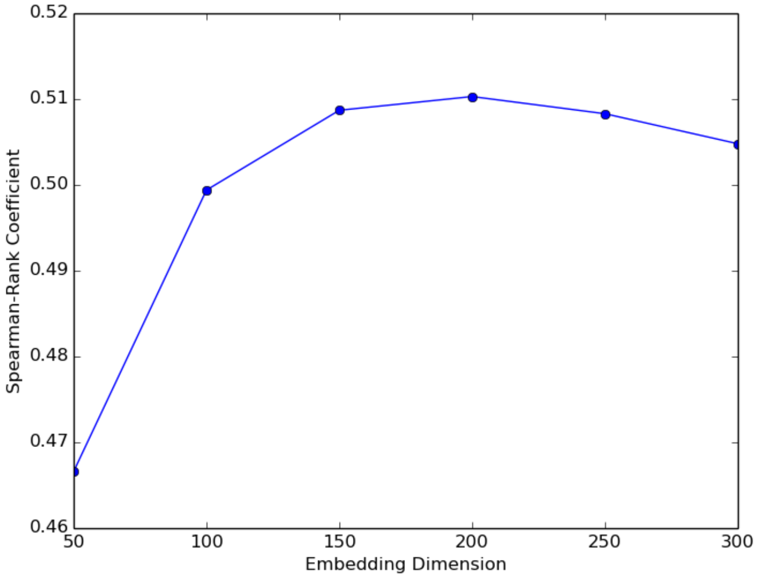


Figure 5.2: Shows the effect of varying embedding dimensionality of our Model on the SCWS task

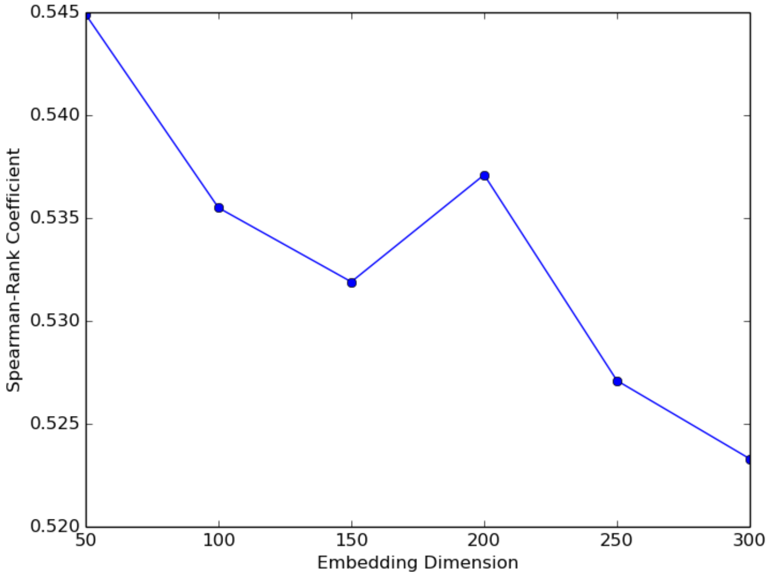


Figure 5.3: Shows the effect of varying embedding dimensionality of our Model on the word353 task

Different Syn1 Property

From 5.5 , it is very obvious that if syn1 has multiple sense embeddings (output embeddings), the final loss is much smaller, although it needs more time to achieve convergence. The reason should be clear somehow. For each center word with giving sense, it has more options to choose, so the final loss is obviously trained smaller. But from the comparison of nearest words in 5.6, we can find that experiment 4 (multiple output embeddings) can not really split up the meaning of word senses, different senses of each word are very similar to each other based on the nearest words. This results also let us think about our model again. Maybe the output embedding should not be prototypes. But we don't have theoretical knowledge to explain such case and for now can not explain it properly. We think this can be the analysis working to do in the future.

id	syn1	t1	t2	iter	t3	t4	loss	SCWS	word353
7	true	849	343	35	1838	64335	0.2457	0.4371	0.4891
11	false	1192	365	45	2866	128949	0.2069		0.4802

Table 5.5: Different Syn1 Property Comparison

	id 7 , one sense output embedding	id 11, multiple senses output embedding
apple	cheap, junk, scrap, advertised chocolate, chicken, cherry, berry macintosh, linux, ibm, amiga	kodak, marketed, nokia, kit portable, mgm, toy, mc marketed, chip, portable, packaging
bank	corporation, banking, banking, hsbc deposit, stake, creditors, concession banks, side, edge, thames	trade, trust, venture, joint trust, corporation, trade, banking banks, border, banks, country
cell	imaging, plasma, neural, sensing lab, coffin, inadvertently, tardis cells, nucleus, membrane, tumor	dna, brain, stem, virus cells, dna, proteins, proteins dna, cells, plasma, fluid

Table 5.6: Nearest words comparison

5.2 Case Analysis

In the following, we will select only one experiment's result to do visualization and nearest words. The selection is based on the final loss and similarity task, specifically it is experiment 7 from above.

Firstly we give the result from *apple*, which is very clear. Different sense has different meanings. Table 5.7 shows the sense similarity matrix of *apple*. The similarity value is the cosine similarity between two embedding vectors. And table 5.8 shows the nearest words of different senses from *apple*. We can see that $apple_0$ and $apple_1$ are about food. They are similar somehow. And $apple_2$ is about company. The next are some sentence examples including *apple* in Table 5.9. These are assigned sentences from the last iteration of training. To make it clear, we only display the sense label of the *apple*. We selected 100 nearest words for each sense of *apple* and do t-SNE embedding to reduce the dimension to 2. And then we only displayed 70% of words randomly to make visualization better, which is shown in Figure 5.4. And we use another table (Table ..) to show the comparison of with other two models (huang and em).

	$apple_0$	$apple_1$	$apple_2$
$apple_0$	1.000000	0.788199	0.800783
$apple_1$	0.788199	1.000000	0.688523
$apple_2$	0.800783	0.688523	1.000000

Table 5.7: Sense Matrix Of *apple*

$apple_0$:	cheap , junk , scrap , advertised , gum , liquor , pizza
$apple_1$:	chocolate, chicken, cherry, berry, cream, pizza, strawberry
$apple_2$:	macintosh, linux, ibm, amiga, atari, commodore, server

Table 5.8: Nearest Words of *apple*

$apple_0$	he can't tell an onion from an $apple_0$ and he's your eye witness some fruits e.g $apple_0$ pear quince will be ground
$apple_1$	the cultivar is not to be confused with the dutch rubens $apple_1$ the rome beauty $apple_1$ was developed by joel gillette
$apple_2$	a list of all $apple_2$ internal and external drives in chronological order the game was made available for the $apple_2$ iphone os mobile platform

Table 5.9: Sentence Examples of *apple*

Next, we select other 5 words *fox* , *net* , *rock* , and *plant*, and also list nearest words as Table .. and sentence examples as Table ... in the following. The example sentences are also cut by ourself without affecting the meaning of the sentence. It's not difficult to find that *fox* has meanings: ; *net* has meanings: ; *rock* has meanings: ; *plant* has meanings: .

In the last, for each sense of each word (*apple*, *fox*,*net*,*rock* and *plant*), we select only 20 nearest words, and combine them together to do another t-SNE embedding, which is also two dimension. The the result is shown in Figure 5.5.

fox	archie, potter, wolfe, hitchcock, conan, burnett, savage buck, housewives, colbert, eastenders, howard, kane, freeze abc, sky, syndicated, cw, network's, ctv, pbs
net	generates, atm, footprint, target, kbit/s, throughput, metering trillion, rs, earnings, turnover, gross, euros, profit jumped, rolled, rebound, ladder, deficit, snapped, whistle
$rock$	echo, surf, memphis, strawberry, clearwater, cliff, sunset r b, hip, roll, indie, ska, indie, hop formations, crust, melting, lava, boulders, granite, dust
run	blair, taft, fraser, monroe, precinct, mayor's, governor's streak, rushing, tying, shutout, inning, wicket, kickoff running, tram, travel, express, trams, inbound, long-distance
$plant$	plants, insect, seeds, seed, pollen, aquatic, organic flowering, orchid, genus, bird, species, plants, butterfly electricity, steel, refinery, refinery, manufacturing, gas, turbine

Table 5.10: Nearest words from fox , net , $rock$, run and $plant$

fox	run by nathaniel mellors dan fox_0 andy cooke and ashley marlowe he can box like a fox_1 he's as dumb as an ox the grand final was replayed on fox sports australia and the fox_2 footy channel
net	net_0 supports several disk image formats partitioning schemes in mr cook was on the forbes with a net_1 worth of billion nothin but net_2 freefall feet into a net below story tower
$rock$	zero nine is a finnish hard $rock_0$ band formed in kuusamo in matt ellis b december is a folk $rock_1$ genre singer-songwriter cabo de natural park is characterised by volcanic $rock_2$ formations
run	dean announced that she intends to run_0 for mayor again in the november election we just couldn't run_1 the ball coach tyrone willingham said the terminal is run_2 by british rail freight company ews
$plant$	these phosphoinositides are also found in $plant_0$ cells with the exception of pip is a genus of flowering $plant_1$ in the malvaceae sensu lato was replaced with a new square-foot light fixture $plant_2$ in sparta tn

Table 5.11: Sentence Examples of fox , net , $rock$, run and $plant$

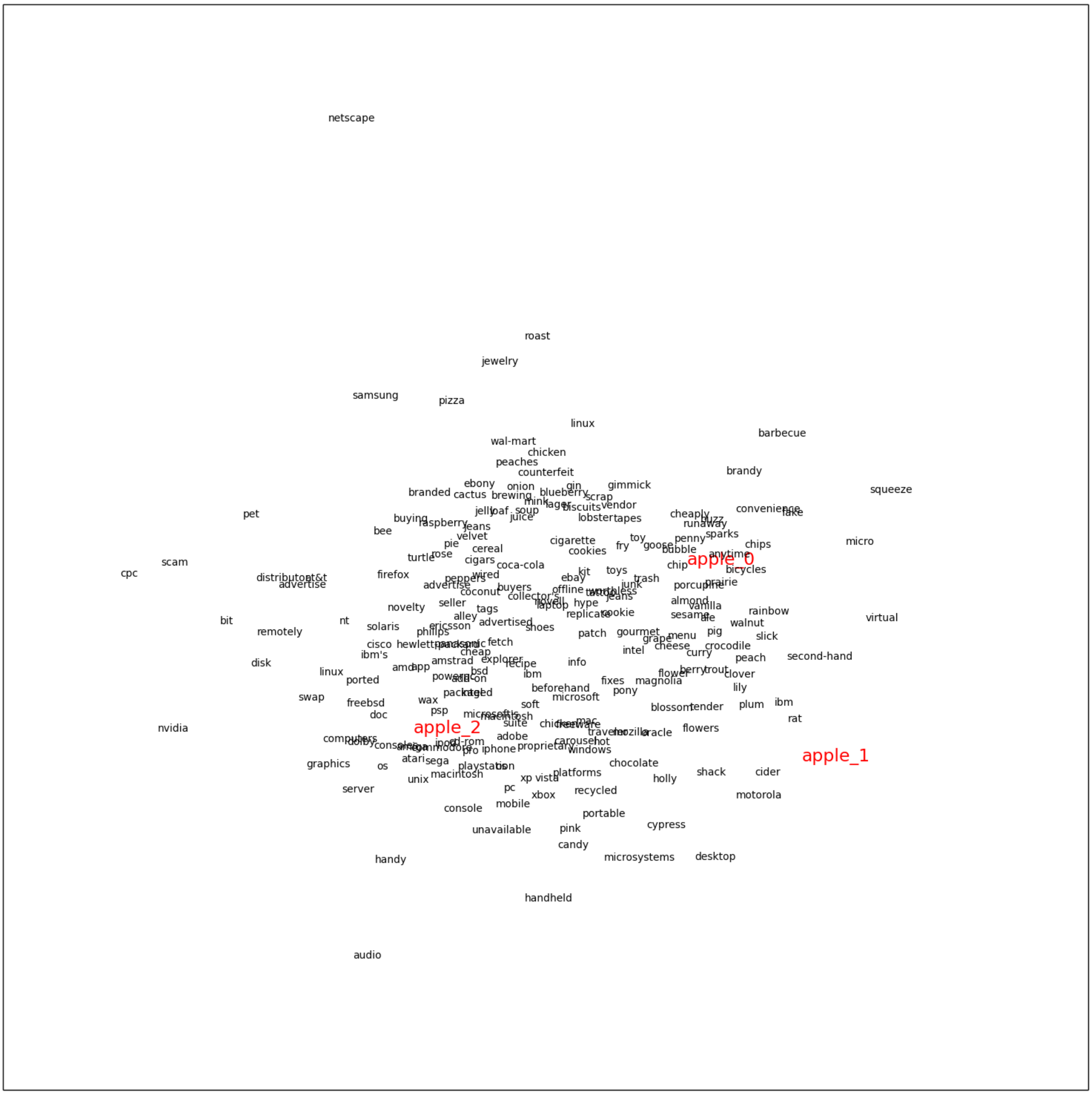


Figure 5.4: Nearest words from *apple*

From the such results, we can say our model is successful to get multiple senses vectors. And it really makes sense.

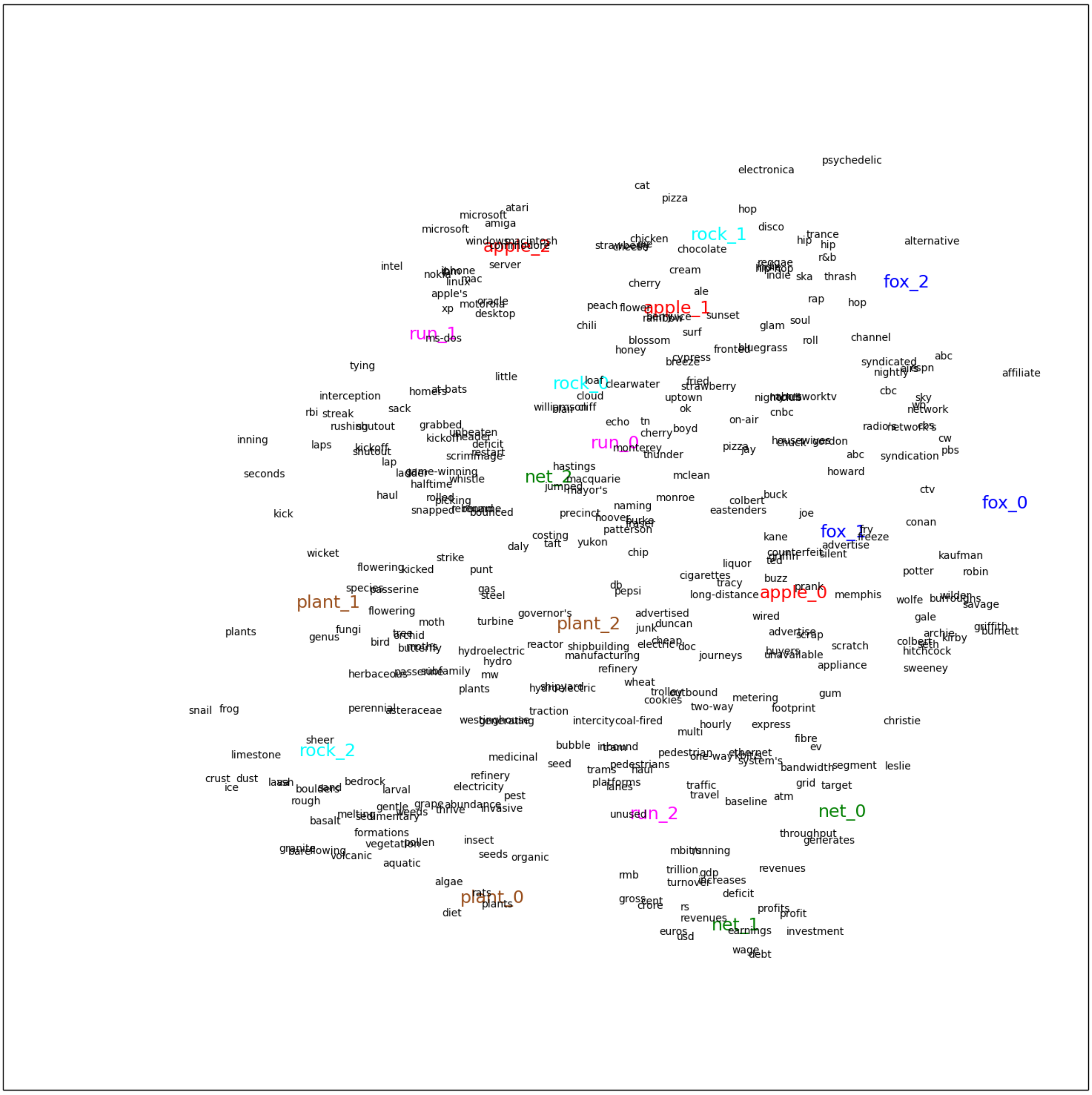


Figure 5.5: Nearest words from *apple*, ,*fox*, ,*net*, ,*rock*, , *run* and *plant*

Chapter 6

Conclusion

In chapter 2, we introduce several word embedding methods and sense embedding methods. Based on these models, we start some new idea and display our model in the chapter 3. After that we compare different experiment any and get some experience about parameters choosing. The most important thing is that originally our model assume that for each word both input embedding vector and output embedding vectors have multiple prototypes (several sense embedding vectors). But the the experiment result is bad. So the we use only one sense for output embedding vectors. After that we got some reasonable results, which achieve our original goal. I think we lack enough theoretical knowledge to support our model's correctness. We just start our working from some ideas. So we really meet many problems in the exploring the methods.

The spark framework is very convenient to use. In the processing of training our word embedding vectors, we gain many turning experience of the techniques. And the experiments showed that our model and implementation is really efficient. Maybe comparing with some c++ implementation, it is not the best choice based on the running time.

However, the evaluation on similarity tasks seems not very satisfied comparing with other models. Maybe in the future, we can do more related working to improve our model. We can try bigger size of embedding vector. Of course, we should in the mean time deal with the memory problem introduced by bigger vector size. On anther hand, we can also do more pre works such as remove the stop words, which may also improve our results.

Appendix A

Appendix

Bibliography

Blei, D. M., Ng, A. Y., and Jordan, M. I. (2003). Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022.

Chen, X., Liu, Z., and Sun, M. (2014). A unified model for word sense representation and disambiguation. In *EMNLP*, pages 1025–1035. Citeseer.

Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., and Kuksa, P. (2011). Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12(Aug):2493–2537.

Collobert, R. and Weston, J. W. (2008). A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning (ICML)*. ACM.

Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., and Harshman, R. (1990). Indexing by latent semantic analysis. *Journal of the American society for information science*, 41(6):391.

Fellbaum, C., editor (1998). *WordNet: An Electronic Lexical Database.* MIT Press, Cambridge, MA.

Harris, Z. S. (1954). Distributional structure. word, 10 (2-3): 146–162. reprinted in fodor, j. a and katz, jj (eds.), readings in the philosophy of language.

Huang, E. H., Socher, R., Manning, C. D., and Ng, A. Y. (2012). Improving word representations via global context and multiple word prototypes. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics: Long Papers-Volume 1*, pages 873–882. Association for Computational Linguistics.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. (2013). Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119.

Neelakantan, A., Shankar, J., Passos, A., and & McCallum, A. (2015). Efficient non-parametric estimation of multiple embeddings per word in vector space. arXiv preprint arXiv:1504.06654.

Pennington, J., Socher, R., and Manning, C. D. (2014). Glove: Global vectors for word representation. In *EMNLP*, volume 14, pages 1532–43.

Salton, G., Wong, A., and Yang, C.-S. (1975). A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620.

Socher, R., Perelygin, A., Wu, J. Y., Chuang, J., Manning, C. D., Ng, A. Y., and Potts, C. (2013). Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the conference on empirical methods in natural language processing (EMNLP)*, volume 1631, page 1642. Citeseer.

Tian, F., Dai, H., Bian, J., Gao, B., Zhang, R., Chen, E., and Liu, T.-Y. (2014). A probabilistic model for learning multi-prototype word embeddings. In *COLING*, pages 151–160.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark: cluster computing with working sets. *HotCloud*, 10:10–10.

Zhou, J. and Xu, W. (2015). End-to-end learning of semantic role labeling using recurrent neural networks. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics.*